



National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

UNIVERSITY OF ALBERTA

Uses of Bounding Boxes In The
Object Modeling Language

BY

Annette B. Gottstein



A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of Master of Science.

DEPARTMENT OF COMPUTING SCIENCE

Edmonton, Alberta
Fall 1994



National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file *Votre référence*

Our file *Notre référence*

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-95037-4

Canada

UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Annette B. Gottstein

TITLE OF THESIS: **Uses of Bounding Boxes In The Object Modeling Language**

DEGREE: Master of Science

YEAR THIS DEGREE GRANTED: 1994

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as hereinbefore provided neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

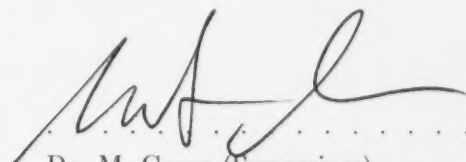
(Signed) *Annette B. Gottstein*
Annette B. Gottstein
RR#3
Lakeside, Ontario
N0M 2G0

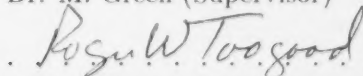
Date: *Aug. 30/94*

UNIVERSITY OF ALBERTA

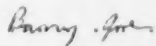
FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Uses of Bounding Boxes In The Object Modeling Language** submitted by Annette B. Gottstein in partial fulfillment of the requirements for the degree of Master of Science.


.....
Dr. M. Green (Supervisor)


.....
Dr. R.W. Toogood (External)


.....
Dr. J. Buchanan (Examiner)

... 
Dr. B. Joe (Examiner)

Dr. M.T. Ozsü (Chair)

Date: *May 30/94*

Abstract

A virtual reality system must balance the need for realistic object interaction, such as collision detection, with the need for a fast, interactive display. The more complex an object's geometry, the more realistic its appearance. However, performing interference detection and prediction on such complex objects is time-consuming and reduces the drawing rate. The drawing rate can be improved by culling those objects that cannot be seen given the user's current position and line of sight.

This thesis focuses on extending the Object Modeling Language to provide collision detection and prediction, and object culling. The approach we take uses bounding boxes to approximate an object's geometry. The box simplification reduces the cost of calculations both in predicting collisions and determining if an object is visible and should thus be drawn. Potentially, all pairs of objects in an environment must be checked for collision. As the number of objects in an environment increases, this becomes an obstacle to the real-time requirements of a virtual reality system. We therefore investigate an approach in subdividing the drawing area in order to reduce the number of object pairs that must be tested.

Acknowledgements

I extend my appreciation to my supervisor, Dr. Mark Green, for his support throughout the course of this work, and to the members of my committee for their evaluation of this thesis.

Thanks also :

to Sean and Chris for all their helpful suggestions, especially those that involved going to RATT, and to Terry for helping me visualize in more than 2-dimensions

to the theorists, Mira and Kathy, for so often tempting me with coffee

to my parents, for their support throughout my academic life

and especially to Ajai for tirelessly telling me I was good enough, and smart enough, and that gosh darnit I better hurry up and get back to Ontario.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Goals	3
1.3	Overview of Thesis	3
2	Previous Work	5
2.1	Collision Detection Techniques	5
2.2	Reducing the Number of Collision Tests	13
2.3	Applicability to Our Work	16
3	Background	18
3.1	Overview of OML and Its Use with MR	18
3.2	Transformations of OML Objects	19
3.3	Creation of Objects in OML	21
4	Calculating the Bounding Boxes	22
4.1	Determining an Approximating Geometry	22
4.2	Calculating the Bounding Box	24
4.3	Determining Object Velocity	25
5	Collision Detection	28
5.1	Checking for Collisions	28
5.2	Improving Collision Detection	29
5.3	Spatial Subdivision	29
5.3.1	Subdividing into Fixed Size Boxes	29
5.3.2	Subdividing into a Fixed Number of Boxes	30
5.3.3	Subdividing into Increasingly Smaller Volumes	30
5.3.4	Mapping Bounding Boxes to the Spatial Grid	31
5.4	Predicting Collisions	32
5.4.1	Checking at Each Time Step	33
5.4.2	An Approximation using Distance	33
5.4.3	A Bisection Sweeping Approach	34
5.4.4	Combining a Sphere Approximation with Sweeping	38
5.5	Extensions for Sub-object Collision Detection	39

6	Culling	40
6.1	Introduction	40
6.1.1	Calculating the View Volume	41
6.2	Culling Techniques Implemented	41
6.2.1	A Sphere Approximation	41
6.2.2	An Approximate Object/View Volume Intersection	46
6.2.3	An Exact Object/View Volume Intersection	46
6.3	Extensions for Culling Sub-Objects	47
6.4	Limitations of These Techniques	47
7	Testing	48
7.1	Evaluating Subdivision in Collision Detection	48
7.1.1	Case of Several Moving Objects, Few Static Objects	49
7.1.2	Case of Few Moving Objects, Several Static Objects	51
7.1.3	Analysis	53
7.2	Comparison of Culling Techniques	53
7.2.1	The First Test Environment	53
7.2.2	Calculating the Number of Objects Culled	54
7.2.3	Calculating the Improvement in Speed	58
7.2.4	The Second Test Environment	58
8	Conclusions and Future Work	64
8.1	Conclusions	64
8.2	Future Work	65
8.2.1	Extensions to OML for Collision Detection	65
8.2.2	Improvements to Collision Detection	66
8.2.3	Improvements to Culling	67
	Bibliography	68

List of Figures

1	Rotation of a Hierarchical OML Object	20
2	Closest Fit of Bounding Volumes	20
3	A Poor Bounding Box Approximation	23
4	Transformation of Bounding Volumes	25
5	90 Degree Rotation of Highest-Level Bounding Box	26
6	Subdividing the Drawing Area into Grid Blocks	32
7	Sweeping a Bounding Box Out Over Time	34
8	Sweeping a Box In The Positive X-Direction Only (as seen from above)	35
9	Sweeping a Box In Both Positive X and Y-Direction (as seen from above)	35
10	Sweeping a Box In Positive X,Y and Z-Direction (as seen from above)	36
11	Sweeping a Box In Positive X and Negative Y	37
12	Difficulty in Choosing Test Points for Culling A Bounding Box	42
13	Eye Position Inside Object	43
14	Centre of Object is in Viewer's Range	43
15	Point on Rim of Object is in Viewer's Range	44
16	Viewer's Range is Within the Object	45
17	Calculating Angles for Sphere-Cone Culling Method	45
18	Graph of Real Time Polygon Rates for Several Moving Objects	50
19	Graph of User Time Polygon Rates for Several Moving Objects	50
20	Graph of Real Time Polygon Rates for Several Stationary Objects	52
21	Graph of User Time Polygon Rates for Several Stationary Objects	52
22	Graph Indicating Real Time Speedup of Culling Techniques for 1 Floor	59
23	Graph Indicating User Time Speedup of Culling Techniques for 1 Floor	59
24	Graph Indicating Real Time Speedup of Culling Techniques for 3 Floors	60
25	Graph Indicating User Time Speedup of Culling Techniques for 3 Floors	61
26	Graph of Real Time Polygon Rates for Culling Techniques	62
27	Graph of User Time Polygon Rates for Culling Techniques	63

List of Tables

7.1	Polygon Rates - Several Moving, Few Stationary Objects	49
7.2	Polygon Rate - Many Stationary, Few Moving Objects	51
7.3	Axis-Aligned Culling Test For 82 Objects	54
7.4	Culling Tests Applied in Order of Complexity For 82 Axis-Aligned Objects	55
7.5	Unaligned Culling Test For 82 Objects	56
7.6	Culling Tests Applied in Order of Complexity For 82 Unaligned Objects	56
7.7	Culling Tests Ordered by Complexity For a Maze of 3 Separated Floors	57
7.8	Timings of Culling For 10 iterations of a 1-Floor Maze	58
7.9	Timings of Culling For 5 iterations of a 3-Floor Maze	60
7.10	Percentage of Polygons Culled	61
7.11	Real-Time Polygon Rate With Various Numbers of Moving Objects .	63
7.12	User-Time Polygon Rate With Various Numbers of Moving Objects .	63

Chapter 1

Introduction

1.1 Motivation

A common criticism of virtual reality is, ironically, the low degree of reality displayed in virtual world applications. Often, interactions between objects are unnatural, if not, in fact, physically impossible, such as the movement of seemingly solid objects through each other. The illusion of reality is further hampered by the poor quality of the images.

While complex objects appear more realistic than simple ones, they are difficult to manipulate. Much time is spent in mathematical calculations to enable natural-looking object interaction, such as collision detection and response. This time factor is critical in allowing realistic interaction between the user and the virtual objects. Generally, the complexity of objects must be sacrificed to achieve reasonable redraw and response rates.

Much of the work in collision detection has been concerned with the exact calculation of a collision time and location. Often, this requires motion to be fully pre-specified, so that the location of an object is known at any point in the future. For virtual reality applications, we cannot always predict the motion of an object, especially since it can be interactively manipulated by the user. Furthermore, mathematically expensive, though exact, collision detection methods prove too

time-consuming for real-time response.

Drawing time can be reduced considerably by culling the objects and the parts of objects that cannot be seen given the viewer's current position and line of sight. Culling has been explored extensively within the field of ray tracing, using techniques that preprocess the objects based on possible viewpoints or on spatial positioning. In ray tracing, the viewpoint is fixed, and objects are generally immobile. In virtual worlds, however, the position of both the viewer and the objects can change, rendering traditional ray tracing techniques inadequate.

The Object Modeling Language (OML) developed by Professor M. Green [10], allows for the creation of complex hierarchical objects to be used in virtual environments. The geometry of these objects can be transformed and changed throughout the execution of an application, allowing the user to create animated objects with interesting behaviours. However, as the complexity of the object increases, more time is required to redraw the virtual scene, and to calculate interactions between pairs of objects.

Currently, detecting collisions between objects in the environment is left for the user to implement. An application in OML can potentially consist of a large number of objects. As the number and complexity of objects increase, it becomes tedious for the designer to manually implement checks for object interference. Encoding a correct collision detection algorithm is made more difficult by the geometrical complexity of OML objects and by the time requirements of an interactive real-time virtual application.

We propose extending OML with commands to allow collision detection, collision prediction and culling of objects not visible given the viewer's current line of sight. To meet the speed requirements of virtual reality, we need to make these methods as efficient as possible. Using the complex, hierarchical representation of an object directly to determine collisions would be too time-consuming.

However, the geometry of an object with a complex hierarchy can be approximated using a bounding box, that is, the smallest 3-dimensional axis-aligned box that can

completely contain the object. If two bounding boxes do not intersect, neither do the objects contained in them. Similarly, if no part of a bounding box is within the range of vision, neither is the object it represents. Using bounding boxes, determining the time until a collision between two objects or whether or not an object should be rendered involve much simpler calculations.

1.2 Goals

Bounding boxes will be used in two manners. First, bounding boxes will be used to maintain information about the various objects within the environment. We wish to detect collisions at both the object and sub-object level. We also want to allow collision avoidance by tracking the time until a collision would occur. This involves maintaining the velocity of an object through its environment. Collision detection should require a minimum of effort on the part of the user. Additionally, it should not slow down the application through excessive computational requirements.

Second, we wish to make rendering as efficient as possible by culling the objects which are not visible given the viewer's current line of sight. Culling can be done at a gross or fine level, that is, culling either the entire object or subparts of it. As well, we do not want to draw objects that are too distant from the viewer to be recognized.

1.3 Overview of Thesis

The thesis begins with an overview of previous work in the area of collision detection. Techniques that attempt to minimize the number of calculations required for detecting collisions are also described. Next, we give a brief background on the Object Modeling Language and the creation and manipulation of OML objects. We then describe our implementation of bounding volumes, how they are calculated and maintained throughout execution of an application, and how they are used to calculate velocity. We describe how these volumes are used for collision detection and

culling at an object level, as well as how these ideas could be extended for use at a sub-object level. Finally, we evaluate the success of the various culling techniques in our tests, and describe ideas for future work in collision detection and culling.

Chapter 2

Previous Work

2.1 Collision Detection Techniques

Collision detection has been studied extensively in areas such as motion planning, robotics, and computer simulation. In virtual worlds, collision detection can be used to give a stronger sense of reality, by preventing objects from passing through each other and by signalling responses to contact between virtual objects.

Collision detection is concerned with determining whether objects will collide, and if so, when and where that collision will occur. Previous work has generally concentrated on three types of collision detection: detection between stationary objects only, which is used, for example, in CAD programs to guarantee that parts of a design do not interfere; detection between a single object moving among several stationary ones, which is used in motion planning; and detection between several potentially moving objects. These last two situations are the ones that are the primary concern in virtual reality.

Boyse [2] defines a collision as occurring when two objects interpenetrate or when faces of adjacent objects coincide. He describes a test system which allows users to move objects within the environment. The system monitors this motion and does not allow objects to be placed so that they interfere with others. Objects in this world are represented as polyhedra to simplify the calculations required in detecting collisions.

Objects consist of faces, in which a face is bounded by edges, and each edge is defined by two points. Objects may be part of a hierarchy, which can be used to create more complex objects.

For interference checking between stationary objects, Boyse first tests the bounding spheres or boxes for intersection, followed by a more calculation intensive, precise intersection test if the bounds appear to intersect. The exact test for intersection is done by tracing rays from points on the edge and from the vertices of one object and counting the number of faces crossed in the second object. If there is an odd number of crossings or if the ray goes from the inside to the outside of the object then the two objects intersect.

For interference checking between a moving and a stationary object, Boyse defines a collision as occurring when the face of one object contacts the face of the other object. He proves that it is sufficient to detect a collision of an edge with a face of the other object. Again, he does an initial test between a bounding cylinder produced by the moving object and the sphere containing the stationary one and does an exact check only if these intersect. For a moving and a stationary object, there are two potential types of collision. In the first, contact occurs between an edge of the moving object and the interior of a face of the stationary object, in which case one endpoint of the edge will be the first to collide with the face. As the endpoint moves over time, it creates a 3-dimensional curve: if this curve intersects any face of the other object, the two objects will eventually interfere. The second type of collision occurs when the edge of the moving object contacts the edge boundary of a face. As the edge moves over time, it creates a surface in space. If this surface intersects the boundary of a face on the stationary object, the two objects will collide. By calculating all collisions and remembering the one occurring earliest in time, this technique can also determine the distance between objects and the time until collision. However, extending Boyse's technique to two moving objects proves inadequate, since simply detecting a collision at some point between the swept out volumes of the moving objects does not guarantee that they occupy that space at the same time.

Hahn [11] describes a hierarchical method of defining objects using bounding boxes, in which collision detection is done at discrete intervals of time. Extending the two methods described by Boyse [2], Hahn detects the place and time of collision by moving objects one time interval, seeing if a collision occurs, and then working backwards in time to determine the first instance of actual interference. One case of object penetration occurs when a vertex of an edge intersects the polygon of another object. A ray is drawn from the collision point along the negative relative velocity of the two objects, and the time at which the objects touch, but do not intersect, is the start time of the collision. The other case occurs when the edge of one object penetrates more than one face of the second object. Again, a ray is traced in the reverse velocity direction to show the path the penetration point took - the earliest time of all these points is the initial time of intersection. As with Boyse, Hahn makes the assumption that the time step used in determining intersection is small compared to the velocity of the polygons. When this is not the case, some collisions may be missed if the objects "jump" over each other within one time frame.

Cameron [3] discusses two approaches to the clash detection problem with respect to robotics. The first samples the motion of the two objects at a set number of time steps, thus potentially using many simpler tests to determine collisions. Cameron addresses the problem of determining an appropriate time step - choosing too small a time step may result in many unnecessary tests, but too large a step may result in missed collisions. He proposes allowing the system to determine the appropriate time step using a divide and conquer approach. For given start and end times, the minimum distance between the objects is found. If a collision occurs, this distance will be less than or equal to 0. If the distance is too far to be reached in these time steps, no collision occurs. If a collision is possible within this time frame, the time is subdivided and each half interval is again checked. This continues until the possibility of a collision is rejected or confirmed. This algorithm performs best if a collision occurs, or if the objects are far apart, which halts the subdividing process.

Cameron's second approach is to test for intersections directly in four dimensions,

which is expanded in his later paper [4]. In this work, he concentrates more on the accuracy of collision detection with respect to a large variety of shapes, rather than efficiency of the algorithm. Each object consists of a set of points and a location function, which gives the position of the object for any point in the future. By extruding objects over time into the fourth dimension, a test for collision consists of merely checking the two extrusions for intersection. These extrusions are built using Constructive Solid Geometry, in which simpler shapes are combined using operations such as union, intersection and difference to produce the desired, more complex, shape. For simplicity, Cameron combines half-spaces to create the desired shapes, since the extrusion into 4-D of a half-space is another half-space. Each expression consisting of a shape and its corresponding motion is converted into a tree, where the leaf nodes are the half spaces used to approximate the object, and the branch nodes are the operations used to combine them. These trees are then extruded into 4-dimensions.

Determining a collision between two extruded trees is done in three stages. First, an approximation to the subtrees is created using S-bounds, which are used to focus the attention of the algorithm. S-bounds describe an area of space by combining subsets of space, such as half-spaces, using intersection and union operators. The advantage is that these subsets of space are easier to describe and manipulate than the resulting space. S-bounds are created by organizing the constraints between half-spaces to allow quick decisions as to which parts of the tree are mutually exclusive. Often, this bound set is rewritten using a smaller number of rules, and entire subtrees may be replaced by the null set which then do not need to be explored further. The second stage uses a divide-and-conquer approach to break the region containing the object into smaller parts. The region, which is axis-aligned, is divided along the axes, and a copy of the object tree associated with it is copied into all subregions. This tree is simplified by removing leaves that correspond to half-spaces that do not pass through this region. The subdivision process continues until the complexity of all regions is deemed to be simple enough to continue onto the third stage of generate and

test. In this last phase, a "sufficient" number of points in space-time are generated and each is checked for intersection in the extruded half-spaces.

Solving the collision detection problem in four dimensions is also the approach taken by Joseph and Platinga [13]. They distinguish between a point in configuration space, which is the location and orientation of an object, and a point in space-time, which is a point in the configuration space at a particular time. Their method is a generalization of Dobkin and Kirkpatrick's 1983 algorithm for determining the polyhedron separation for convex 4-prisms. This method finds and minimizes the separation of every pair of faces in 4-space, and so determines the time of the first collision.

To create the 4-prism, copies of the polyhedron P are made at the starting and the ending time points. The end vertices of these copies are connected with an edge, the edges of the copies are connected with a 2-D face, and the faces of the two copies are connected with a 3-D face. The same is done with the second polyhedron, Q . Next, hierarchical representations of these 4-prisms are created, $P_1, \dots, P_r$ and $Q_1, \dots, Q_s$, where each successive approximation contains fewer vertices than the last, and where $P_1 = \text{the 4-prism for } P$. Each approximation is formed by removing an independent set of vertices and taking the convex hull of the rest. For example, take level i to be the lesser of the 4-prisms hierarchical levels r and s . The points p and q which represent the closest points between P_i and Q_i are calculated. The vertices that make up this representation are removed, giving level $(i - 1)$, and the process is repeated, until $(i = 1)$. The resulting points p and q are the points in the polyhedra P and Q which produce the minimum separation in 4-dimensions.

Culley and Kempf [6] take the approach that for applications such as path planning, an exact collision test is not required. It is adequate to find a "safe" time interval in which no collision can occur, even if that interval is smaller than the exact one. Thus, a conservative approximation of the objects can be used to simplify tests for collision. The idea behind their method is that by determining the minimum distance between two objects, and the maximum speed of both, one can calculate the earliest time for a collision. For two bounding spheres, this distance is simply

the Euclidean distance between the centres minus the sum of the radii. Once this conservative time is calculated, the exact time of contact can be found by advancing the simulation by small time steps, and backing up if object penetration rather than mere contact occurs.

Moore and Wilhelms [16] conclude that computationally expensive general solutions to collision detection are not required in animation systems; instead, the speed with which collisions are detected is of greater importance. They present two algorithms, dealing with flexible surfaces and convex polyhedra respectively. Moore and Wilhelms model flexible surfaces as a grid of points connected into triangles. In this representation, collision detection consists of checking if a point penetrates any of the triangles to which it does not belong. At each step, the position of the point at the start is compared to the position of the point at the end, specifically, the perpendicular distance from the point to the triangle is calculated. If the sign of the distance changes, the point crosses the triangle in this time step, and the exact point can be found by solving parametric equations.

Their technique for collision detection between convex polyhedra is based on the Cyrus-Beck clipping algorithm. By taking the dot product of a face's outward pointing normal with a vector on the edge, it can be determined on which side of the face the point lies. If a point lies towards the inside of all six planes represented by the faces of the polyhedron, the point intersects the polyhedron.

There are three possible intersection tests :

- a vertex in B intersects a face of A
- an edge of B intersects a face of A
- A and B are identical and moving through each other.

If any point is found to be inside the other polyhedron, the algorithm can terminate. First, the vertices of B are checked for inclusion in A using the method described above. Next, the edges of B are checked for intersection against the faces of A . If the perpendicular distance from the edge's endpoints to the face's plane differ in sign, the

edge intersects the plane, and the intersection points can be found using the following equations:

$$\begin{aligned}d_i &= (v_i - u_{k1}) \cdot n_k \\d_j &= (v_j - u_{k1}) \cdot n_k \\t &= \frac{|d_i|}{|d_i| + |d_j|} \\ \text{giving the point : } P &= v_i + t(v_j - v_i).\end{aligned}$$

where $v_i v_j$ represents the edge, u_{k1} is a vertex on the face, n_k is the normal of the face, and d_i and d_j are the distances from the endpoints to the plane. Evaluating these equations for all plane faces of polyhedra A give a series of t values between 0 and 1, which are kept in sorted order. Each pair of points resulting from these t values, including the endpoints, represents a segment of the edge which crosses one of the face planes, but which does not necessarily cross it within the polyhedra. The midpoint of each of these segments is checked for inclusion in polyhedron A , using the same method as in the first check. Lastly, the centroid of each face is tested for inclusion in A , in case the faces are perfectly aligned and moving through each other. The tests are then repeated, checking for the inclusion of points of A inside B .

Collision detection using this method is $O(n^2 m^2)$ for n polyhedra and m vertices/polyhedron. As with other methods, the algorithm will not detect a collision if the objects move completely through each other in one time step.

Hubbard [12] addresses the fact that virtual reality applications require a collision detection algorithm that is fast enough for use in an interactive environment. This method must be real-time, even when applied to several objects, and must be able to handle objects for which the motion is not pre-defined, since the motion may be directed interactively by the user. He proposes an approximate collision detection technique with progressive refinements, in which the level of collision detection is refined until the process is interrupted by the application. This provides a flexible system in which the quality of detection can be degraded when speed is crucial.

In the first phase of testing, bounding boxes of all objects are compared to determine the earliest possible collision time. No more work is required until that time is reached, at which point only the objects involved in the collision must be processed. In this narrow processing phase, Hubbard creates 4-dimensional space-time bounds to estimate the location of an object in the future. Using the analogy of a point A moving over time, Hubbard shows that a 3-dimensional bound on the position of A at a particular time t is a sphere with a radius of $(M/2)t^2$, where M is a scalar upper bound on the acceleration. This sphere is centred at $x(0) + velocity * time$. Applying this over time to four dimensions produces a series of these spheres, where at any particular time t , the cross-section is a 3-D sphere. This series of spheres resembles a parabolic horn, in which the radius becomes larger with an increase in time t . Extending this idea to non-points creates a union of such 4-dimensional parabolic horns. To simplify calculations, each parabolic horn is bounded with a hyper-trapezoid, consisting of six 4-D faces, (one face for each 3-D face of the original cube). A cross-section of the 4-D face at any time t is an axis-aligned 3-D square, and these cross sections increase in time with t . Intersection between these space-time bounds can be reduced to searching for an intersection between faces. These faces are projected onto a 2-dimensional plane, which produces a line segment. If the two line segments intersect at a time t_i , the cross-segments at t_i must also be checked using linear programming techniques. If these intersect, so do the space-time bounds.

These space-time bounds expire when the upper bound on acceleration (which determines the size of the cross-section) is no longer valid. At this point, the space-time bounds must be recalculated. This approach is most efficient if the time between recalculations is large.

For further refinement, the objects themselves are modelled as a hierarchy of spheres, called **sphere-trees**, where each deeper level uses a larger set of spheres to more closely approximate the geometry of the object. It is from these spheres that the space-time bounds are derived. The quality of collision detection improves with each additional level descended. The application can interrupt the detection algorithm at

any level, thus limiting the accuracy of the detection and the time spent on it.

Hubbard claims that these trees must be built only once per object, and that the spheres can be transformed in the same manner as the object. However, this is true only if the geometry of the object does not change throughout the application, otherwise, the tree must be rebuilt. This collision detection scheme will not work well if many geometry changes occur, requiring the trees to be rebuilt, or if there are many changes to the type of motion, requiring many recalculations of the space-time bounds.

2.2 Reducing the Number of Collision Tests

Many of the methods described above would prove cumbersome in an environment consisting of several moving objects, in which we need to detect collisions between all possible pairs. In fact, the algorithms presented could require $O(N^2)$ object collision checks, and even more if we wish to find collisions between sub-objects. This is wasteful in large environments in which some objects will likely never meet. Other approaches in collision detection have attempted to reduce the number of checks that are required by first determining a smaller region in which collisions may occur for an object, and by building a list of highly likely interfering pairs of objects.

Such an approach is presented by Youn and Wohn [17], who are concerned with finding collisions with sub-objects in a hierarchical representation of an object. They localize possible collision regions by using a tree structure, called **C-trees**, to represent the structure of their objects. Each node of this tree is a bounding shape enclosing all of the children below it. Each leaf is the actual sub-object shape. These nodes localize the region that must be checked - if a collision does not occur with the parent node, it cannot occur with any of the children.

They propose a two-pass algorithm, in which the first step creates a list of leaf-node pairs which are highly likely to collide, and the second step determines collisions

between the pairs in the list. The first pass recursively localizes the region being checked. If two non-leaf nodes intersect, their children are localized against each other. If one node is a leaf and the other a non-leaf, then the children of the non-leaf node are localized against the leaf node. Once we are down to a comparison of two leaves (which represent the shape of the actual sub-object), the pair is inserted into the list of candidates for collision.

This approach can be extended to subdivide the region on an object-level as well, where the leaves of the tree would represent one object structure. Thus, only likely pairs of objects would be checked for intersection on a sub-object basis.

In this approach, much time may be needed to update the C-trees, which must be modified whenever a segment of the object moves. This becomes an important issue in virtual environments with continuously moving objects. Additionally, while it is clear that the configuration of the C-tree determines the efficiency of the collision checking, the authors do not discuss how best to generate these structures. If left to the designer of the virtual world, the optimal configuration cannot be guaranteed.

Dworkin and Zeltzer [7] also use a two-pass system to reduce the number of collision calculations. Recognizing the cost of updating a hierarchical representation of space at each time step in a dynamic environment, they propose using a sparse dynamics model to predict the next collision for each object. This model assumes that the number of collisions is considerably smaller than the number of objects in the environment. Once we have found the first collision for an object, it must only be compared to all other objects once the collision takes place, when a new target must be found. Thus, if few collisions occur, few updates need to be made.

A meeting is predicted by solving the following quadratic formula,

$$t = \frac{-\Delta P \cdot \Delta V \pm \sqrt{d}}{\Delta V \cdot \Delta V}$$

where

$$d = (\Delta P \cdot \Delta V)^2 - \Delta V \cdot \Delta V (\Delta P \cdot \Delta P - (R_1 + R_2)^2)$$

where the positions of the objects are given by $(P_1 + V_1 t)$ and $(P_2 + V_2 t)$, and the radii of the bounding spheres are R_1 and R_2 respectively. This gives the time when

the distance between spheres bounding the objects is less than the sum of their radii. If the discriminant is negative, the two spheres never meet. The smaller root gives the time when the objects meet, whereas the larger root indicates the time when they move apart.

Each element keeps track of its next collision. All possible collisions are maintained in a time-ordered queue. When the time of the first event is reached, an accurate collision detection technique is used to determine if an actual collision occurs. If it does, both objects involved recalculate their next target and place that information on the ordered queue. Finding the first target for each object will require $O(N^2)$ comparisons, but can be done in preprocessing. Updating the target for one object requires $O(N)$ comparisons.

Collision predictions for an object may become invalid, either because the target is hit by something and so is no longer in the object's path, or because the object itself is hit by something and so changes its direction. In the first case, the object must locate a new target and enqueue the collision time. In the second case, the event on the queue is simply ignored, since the path of the object has changed.

Success of this approach depends on the fact that the direction and speed of objects do not change until after a collision occurs. If these factors change for an object, the nearest target and the time of collision have to be recalculated. This would add extra processing time, and potentially leave many stale events on the queue.

Zyda et.al. [18] achieve great efficiency in their animation system by exploiting their knowledge of the objects in the environment. The system presented, NPSNET, displays vehicle movement over ground and through air, which can be controlled either interactively or via a script file. Collisions are detected among pairs of vehicles and between vehicles and stationary terrain features. The world is divided into constant size grids, since the size and the features of the terrain are fixed. Each grid square contains a list of fixed objects that occur in it - for simplicity, the centre of the object determines to which grid it belongs (for many applications, however, this is a bad estimation, since collisions may occur with parts of objects overlapping into other

grids). Whenever an object moves, it is checked for collision against other objects within that grid. In NPSNET, fixed objects are attached to the ground and have a limited elevation - thus, collisions are checked only if the moving object is low enough to the ground to permit interference. Collisions with other moving vehicles are determined by first calculating whether spheres bounding the vehicles intersect, and then using ray tracing techniques to find the actual point of collision, which is needed only for collision response. The assumption made in this technique is that vehicles have a "reasonable" speed, and so will move across only one grid between tests, thus ensuring that all collisions are found.

Although NPSNET achieves real-time performance, it comes at the expense of generality. In NPSNET, collisions outside of the viewer's range are not calculated, since it is assumed that they are of no interest. In a virtual reality system, however, all collisions must be considered, because the line of sight may change, and we would expect that part of the world to have changed as well. As well, collisions trigger responses, and so might change the motion of an object, bringing it back into view. In virtual environments, we cannot make assumptions about the size of the world, the number of objects or their behaviour. Thus an application-specific approach such as NPSNET is inappropriate for our work.

2.3 Applicability to Our Work

Most of the techniques discussed above are concerned with detecting collisions exactly. This level of precision is unnecessary for most of our applications, and is, in fact, undesirable as a default collision detection method. Collision detection may not be required for all applications, so any added geometry overhead that is required to simplify collision detection (for example, bounding boxes) must be minimal. Similarly, the time required to detect collisions between two objects cannot be so long as to miss other collisions. In order to maintain a realistic feel to the virtual reality application, the calculations required to determine collisions (or misses) cannot slow down the

display. Computationally expensive collision detection schemes as described above would be too slow for most virtual reality applications, especially ones in which many objects must be checked.

Chapter 3

Background

3.1 Overview of OML and Its Use with MR

OML (Object Modeling Language) is a procedural programming language used to specify the geometry and behaviour of three-dimensional objects used in virtual worlds. The geometry of objects created through OML is defined independently of specific graphics packages. Interfaces currently exist for the graphics languages GL and PHIGS.

OML allows objects to be specified in a hierarchical manner, providing a means of creating complex objects from relatively simple shapes such as spheres, rectangles and cylinders. An object is the result of a converted (hierarchical) solid, where a **solid** can consist of zero or more **polygons**, as well as zero or more other solids. **Transformations** and **colour** can be applied at all levels within the hierarchy, both to solids and to polygons. These transformations will be applied to the children of the solid/polygon as well. **Labels** and **tags** can be attached to any part of the hierarchy to reference either a certain piece of the geometry or a certain type of sub-object, respectively.

A virtual environment can contain several **instances** of the same object. OML objects may be parameterized, allowing each instance to have unique characteristics, such as a distinct geometry, colour or velocity. Different instances may respond to

different **events**, which may be generated by either the system (for example, clock ticks) or the application (such as a hand gesture, or the intersection of objects). The conditions that generate events are reevaluated every time step. An event may cause a certain **behaviour** to be activated for an instance of an object. These behaviours, specified when defining the object in OML, are used to animate the object. Behaviours can consist of transforming an object or parts of that object, or of changing aspects of its geometry, such as its colour or its modeling hierarchy. If no behaviour is specified for an object, it cannot respond to any events.

The **MR** (Minimal Reality) Toolkit provides an interface between the viewer and the virtual world. It maintains both the viewer's position and their line-of-sight through use of a Polhemus Isotrak device, thus determining which objects are currently visible.

3.2 Transformations of OML Objects

Transformations can be applied to either the whole object, or to parts of the object. Since objects are defined hierarchically, transformations may be applied individually to a child node. If two objects are combined together into one solid and a rotation is applied to them, the two objects will rotate about their individual origins, NOT the origin of the combined object, as shown in Figure 1. This is an important distinction when calculating bounding volumes. We wish to maintain the smallest bounding volume possible throughout the application of any transformations. If we apply the transformation only to the bounding volume at the highest level, it may no longer provide the closest fit to the transformed objects underneath it (see Figure 2).

Translations, rotations and scaling are not commutative. Thus, the order in which the transformations are applied must be maintained if we wish to use these transformations to calculate the new bounding volumes. In OML, the transformations for each solid and polygon are stored in a linked list, with new transformations added to the end of the list.

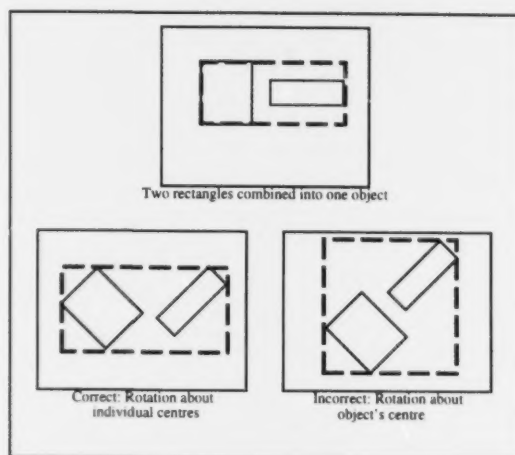


Figure 1: Rotation of a Hierarchical OML Object

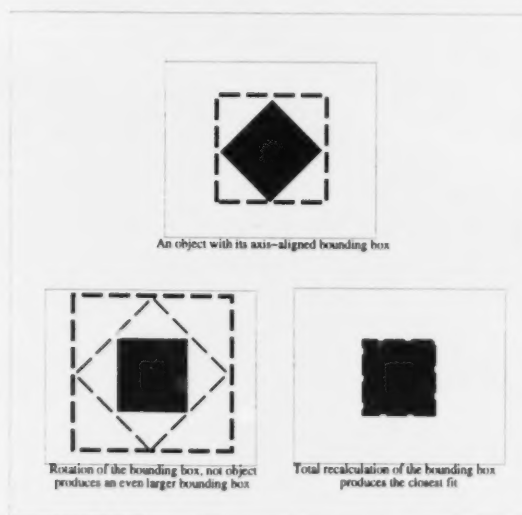


Figure 2: Closest Fit of Bounding Volumes

3.3 Creation of Objects in OML

Several primitives are available to OML users, including rectangles, boxes and spheres. Each primitive consists of a set of vertices and is initially axis-aligned. The exception to this is a polygon created from a path. A path allows the user to state specifically the values of the vertices, thus, axis-alignment is not guaranteed. An object can be transformed (translated, rotated or scaled) only after being made into a polygon (the result of calling any of the routines to create a primitive).

Chapter 4

Calculating the Bounding Boxes

4.1 Determining an Approximating Geometry

There are two main issues to consider when choosing an approximating geometry for three-dimensional objects. First, the approximation must be fairly true to the original form - having objects with approximations that are too generous could cause collisions to be falsely detected, simply because the approximations collide. Alternatively, actual collisions might go undetected if the approximation does not fully contain the object. Second, the approximation must be easily and quickly updated, since objects within virtual worlds can be continuously transformed by their behaviours throughout execution of the application. Thus, savings gained by manipulating the approximations to the geometry must outweigh the cost of maintaining them.

Bounding boxes and bounding spheres are two possibilities for approximating object geometry. Bounding spheres have the advantage of making some calculations easier, since every point on the outside of the sphere is an equal distance from the centre. However, bounding spheres are a poor approximation for objects that are much larger in one dimension than in another, such as walls or floors. Conversely, bounding boxes provide a poor approximation to large spherical objects, in which much empty space will be included in the corners of the box. However, since many

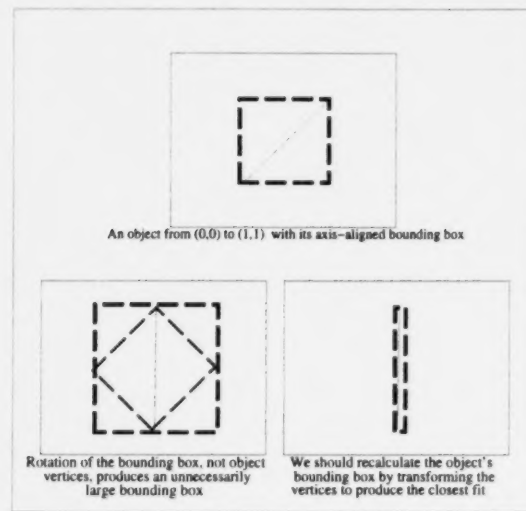


Figure 3: A Poor Bounding Box Approximation

virtual environments tend to include objects such as floors and walls, bounding boxes are used as the primary approximation, from which sphere approximations can be calculated if required.

Bounding boxes can be either axis-aligned or non-aligned. Axis-aligned boxes allow for quick intersection testing, since all that is required is checking the minimum and maximum values in each dimension, resulting in at most six comparisons. Unfortunately, an axis-aligned bounding box is not the best choice for all objects. For instance, an approximation of a line from (0,0,0) to (1,1,1) will have much wasted space in the axis-aligned bounding box representation, as shown in Figure 3. Thus, non-aligned boxes may provide a better fit to the actual object, however, determining and maintaining bounding box information and testing for intersections are more difficult and time consuming than for axis-aligned bounding boxes. As noted in Section 3.3, most objects in OML are created in an axis-aligned manner, making the stored bounding box (the one used in future calculations) a fairly good approxima-

tion.

The information required for each bounding box is added to the structure for both polygons and solids. Thus, bounding boxes themselves are hierarchical, with the bounding box for a solid being the box that contains the transformed bounding boxes underneath it in the hierarchy. A bounding box consists of two OML-vectors (x , y , and z values) representing the maximum and minimum corners of the bounding box. The transformations are stored in a 4×4 matrix, again at each level in the hierarchy. Thus, determining the transformed bounding box requires a multiplication between a 4×4 matrix and the minimum and maximum corner coordinates to find the new minimum and maximum values. This could be done interactively each time the transformed bounding box values are required, however, determining the transformed box once and storing it in the structure allows it to be quickly referenced in several different instances. A direction vector is also stored for each solid, indicating the change in the x , y , and z positions from the last time step.

4.2 Calculating the Bounding Box

When a new polygon is created its list of vertices is searched to determine the minimum and maximum values in each dimension. These values are stored in the polygon's structure. If a new polygon is added to an existing one the bounding box values are compared to determine the maxima and minima of the entire group of polygons. Calculating the bound for a solid requires the comparison of the transformed bounds of the solids and polygons underneath it.

The change of bounding box values at any level in the hierarchy must be reflected in the levels above it (bounding boxes could change through the addition of new sub-objects, or through the transformation of existing sub-objects). Transformations at any level must also be propagated throughout lower levels in the hierarchy, resulting in recalculations of the bounding boxes both beneath and above the transformed level. It was initially thought that some recalculations could be saved by applying

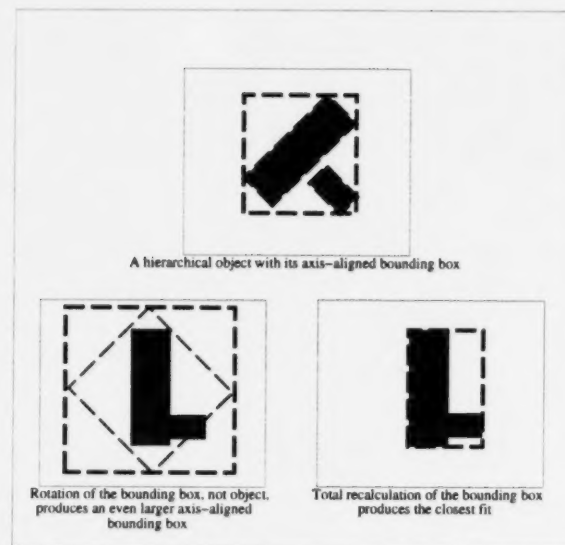


Figure 4: Transformation of Bounding Volumes

the transformation only at the transformed solid's level, however this proved to be a naive assumption. For example, consider the case of a rotation applied to a solid which consists of several sub-objects. When actually drawing the solid, the rotation is applied to each of the sub-objects individually; that is, the sub-objects rotate around their individual centres, not the centre of the solid into which they have been combined. Thus, the new bounding box could actually be smaller than the original - however, by applying the transformation at the parent's level, the new axis-aligned bounding box is even larger than the original, as in Figure 4. A transformation to only the top level bounding box of a hierarchical object could even produce an incorrect, that is, too small, bounding box, as shown in Figure 5.

4.3 Determining Object Velocity

First, we must consider which motion generates a velocity value - obviously, trans-

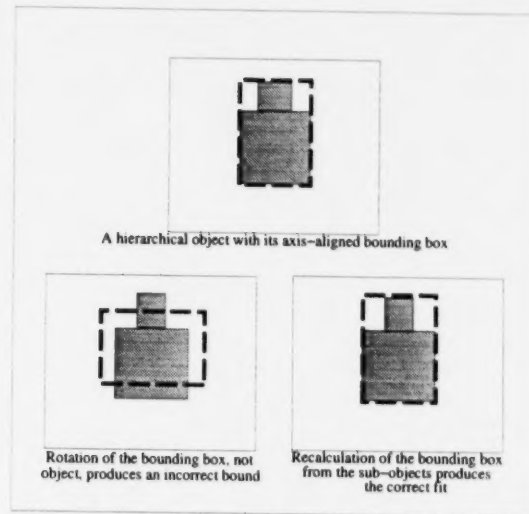


Figure 5: 90 Degree Rotation of Highest-Level Bounding Box

lation does, but what about rotation or scaling? The object may not actually “move” but the points that make up the object *do* move.

Each time the bounding box values change, a new velocity vector must be calculated. Currently, we estimate velocity using the changed position of the centre of the bounding box. This, however, may not be an accurate reflection of an object’s average velocity. Consider an object in which the base remains stationary, but the top grows and shrinks in one-dimension. Its velocity could be considered 0 - it is not actually moving. The current method, however, calculates first a positive velocity (in the direction of growth) and then a negative one, since the centroid of the object moves. However, an object that grows equally in both a positive and negative direction, would have a 0 velocity, since the centroid would remain fixed. As a second example, consider a bird flying, in which the wings flap up and down, but the main motion is forward. The velocity, as calculated using the current method, maps a jagged path instead of a straight one, as the centroid of the bird moves up and down along with

the rise and fall of its wings. In this particular case, calculating the velocity over two time steps instead of one would remedy this problem. But this too would prove inadequate when the bird turns suddenly. It seems any velocity we determine using the bounding box would provide a rough approximation at best. Perhaps allowing advanced users to specify the number of time steps used in determining the velocity would provide more flexibility, meeting the needs of a larger set of applications.

We choose to calculate velocity only at the highest object level. However, each sub-object in the hierarchy could potentially calculate its own velocity in a manner similar to that used for the whole object, if a need for this arises.

Chapter 5

Collision Detection

5.1 Checking for Collisions

A collision may be detected at either a gross (object) or detailed (object subparts) level. The user can request an event to be signalled whenever a collision occurs between two specified items, whether these items are objects, object subparts or a mixture. Each such user request causes a new record to be added to an OML internal list which maintains the pairs of items to check for collision. The stored record consists of the addresses of the object instances and the desired subparts (if required) and the event to be triggered upon collision. To allow collision prediction, a time value is also stored indicating the number of ticks /it in advance /rm of a collision that the event should be triggered. The velocity maintained for each object allows the time until a collision to be estimated.

At the end of each time step this internally maintained list is traversed. The transformed bounding boxes corresponding to each item in a collision-pair are checked for intersection. Since the bounding boxes are axis-aligned, intersection testing requires a maximum of six floating point comparisons to see if an overlap exists in all three dimensions. If an intersection occurs, the event registered with that collision-pair is triggered.

5.2 Improving Collision Detection

The collision detection scheme, as described, depends on the fact that the number of pairs we need to check is quite small. If, however, we want to check for *any* collisions in the environment, it would become tedious to specify all possible pairs, and time-consuming to check all of them at every time step - we may require as many as $O(N^2)$ collision checks.

5.3 Spatial Subdivision

To remedy this, a three-dimensional stationary grid is calculated at the start of the application, which subdivides the entire world drawing space into smaller boxes. Each instance maintains a bit-flag (a 32-bit unsigned long int) which maps to the drawing area. One individual bit indicates whether or not the instance occupies that 3-D space. Obviously, pairs that are physically nowhere near each other cannot intersect.

Choosing a subdivision scheme is more complicated than it would appear, simply because any assumptions made to choose a method can be shown to be inadequate for some hypothetical virtual world that *might* be designed. This is demonstrated in the following subsections.

5.3.1 Subdividing into Fixed Size Boxes

We could choose to make the subdivisions of a fixed size - for instance, 1 cubic metre. This would produce a linked list of flags, giving a total of $(\#subdivisions/32)$ flags. Collision detection would then consist of looping through all the flags, ANDing them until a hit (a nonzero AND value) is made. This could fail in two ways: there may not be enough memory available for a linked list of flags for all instances, or the division size chosen may not be appropriate for all worlds. As an example, suppose someone created a virtual world consisting of the universe - using 1 cubic metre

divisions would surely cause memory allocation to fail. However, increasing the size of the divisions is an inadequate solution. Consider a world consisting of a bee hive...many collisions may occur inside the hive, but all of these would fall within the 1 cubic metre subdivision, thus there would be no savings on the $O(N^2)$ required collision checks.

5.3.2 Subdividing into a Fixed Number of Boxes

Instead of subdividing into a standard grid size, we could choose to subdivide into a standard number of boxes. Thus, the size of the grids would be flexible for each application, but fixed within one application. However, this might not improve the number of required collision checks, if most of the objects fall within the same grid location.

5.3.3 Subdividing into Increasingly Smaller Volumes

The method we chose to implement repeatedly subdivides an area in which a collision has been detected into 32-piece grids (the number that can be represented by one unsigned long integer). If the AND of the objects' bit-flags is 0, we can assume that no collision occurred. Similarly, if the objects intersect in ALL sub-areas, we can assume that a collision is quite likely, and proceed to the exact intersection test (discussed in the previous section). Otherwise, we subdivide further only those sections in which the objects may overlap (those which produced a "1" from ANDing the flags).

The problem with this method arises in knowing when to stop subdividing. It may not prove to be efficient to stop only when the objects intersect in *all* or *none* of the grid sections. This leads to a similar discussion as to which grid size to choose: any fixed size, or fixed number of subdivisions, may be inadequate given a world in which all objects are very small or very large.

Currently, to choose the depth of subdivisions, we calculate the average bounding box size (in x, y and z dimensions) over all object instances. The subdivision depth

in each of the x, y and z dimensions is the number of times we need to divide a grid block by 4 (or 2, in the z-dimension) to obtain a block that is no larger than the average bounding box length in that dimension. We then take the largest of these three depths as our overall subdivision depth. This is demonstrated in the following section. Since we want to maintain integer values for the number and size of grid sections, we do not allow a block size of less than 1. The depth is calculated only once in preprocessing.

5.3.4 Mapping Bounding Boxes to the Spatial Grid

Before beginning the simulation, we traverse all instances and compare their bounds to find the current world size (the minimum and maximum in each dimension, over all bounding boxes). To get 32 grid divisions (to allow storage in a 32 bit integer), we choose to divide the x and y dimensions into 4 pieces each, and the z-dimension into 2 pieces. These pieces will have different lengths, depending on the dimensions of the entire world area. To find the required number of subdivisions along x, we recursively divide a grid box's x-length into 4 pieces until the resulting size is less than the average x-dimension over all bounding boxes. For example, consider a grid size along x of 32 metres, with the average bounding box having an x-dimension of 2 metres. Since the x-dimension gets split into 4 pieces, we would need a subdivision depth of 2:

$$\begin{aligned} 32 \text{ metres} / 4 \text{ pieces} &= 8 \text{ metres/piece} \\ 8 \text{ metres} / 4 \text{ pieces} &= 2 \text{ metres/piece} \end{aligned}$$

Every grid division that is intersected by a bounding box is marked in the bit flag. The starting and ending blocks in x, y and z are determined by dividing the bounding box by the grid step size in each dimension. For example, the values in the x-dimension would be:

$$\begin{aligned} \text{start} \rightarrow x &= (\text{bound} \rightarrow \text{min}.x - \text{drawarea.min}.x) / \text{step}.x \\ \text{end} \rightarrow x &= (\text{bound} \rightarrow \text{max}.x - \text{drawarea.min}.x) / \text{step}.x \end{aligned}$$

If a boundary value is less than 0, it is mapped to 0. Similarly, the upper limit on

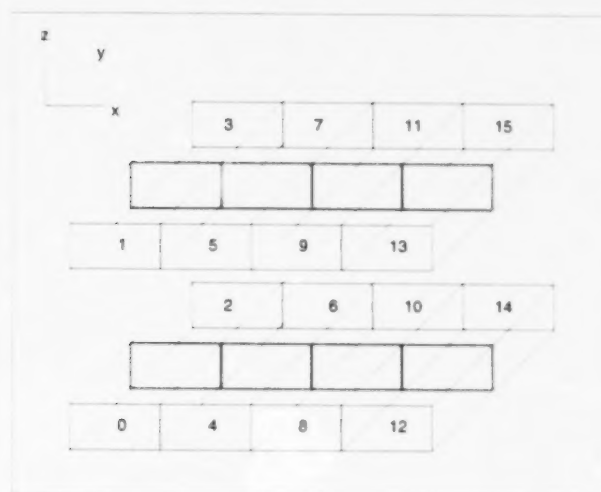


Figure 6: Subdividing the Drawing Area into Grid Blocks

the boundary value is the maximum number of divisions in the given dimension.

Once these x, y and z boundaries are found, they are combined to give a unique grid block. Specifically, blocks are incremented first in the z-dimension, followed by y and then x. This is illustrated in Figure 6 for 4 divisions along the x-dimension, and 2 divisions along the y and z-dimensions.

5.4 Predicting Collisions

In many applications, simply noting a collision once it has occurred is inadequate; often, we want to be able to predict collisions in advance in order to avoid them. For prediction, the user can specify that an event should be signalled a certain number of seconds or system time ticks before the collision. If a time of 0 is specified, the event will be signalled only upon actual collision.

The current implementation checks for a direct collision before checking for a collision within the specified prediction time. This reduces the number of calculations

required, since the intersection of bounding boxes is considerably cheaper than any of the prediction methods. It also has the added advantage that different behaviours could be applied to an object when a collision is predicted than when it actually occurs.

As noted in many of the papers described in Chapter 2, collision prediction is difficult. We have implemented three methods of varying complexity and accuracy to address this issue.

5.4.1 Checking at Each Time Step

If prediction is required for a small number of time units, and if the objects do not move quickly, checking at discrete time intervals can be an effective and low cost method. We simply calculate new bounding box values at each of the time steps until a collision occurs, or until the time increment exceeds the required prediction time.

This technique could miss collisions, however, if the objects move quickly in relation to each other, or if one of the objects is quite thin.

5.4.2 An Approximation using Distance

We estimate the likelihood of a collision between two objects by calculating the closest distance between the 3-dimensional vectors representing their respective velocities, where the base of the vector is the current centre of the object. If this distance is less than the distance between the objects, and if the objects reach this point at approximately the same time, then a collision will occur.

Taking the approach of Dworkin and Zeltzer [7], the distance between two spheres is simply the distance between their centres, minus the sum of their radii. In our application, we can create bounding spheres by using the diagonal distance from the minimum to the maximum points of the bounding box as the sphere's diameter. From these, we can calculate the time at which the spheres first intersect and the time at which they move apart. While this is simple to calculate, it suffers from inaccuracy when modeling objects such as walls, as discussed earlier in Section 4.1, which will

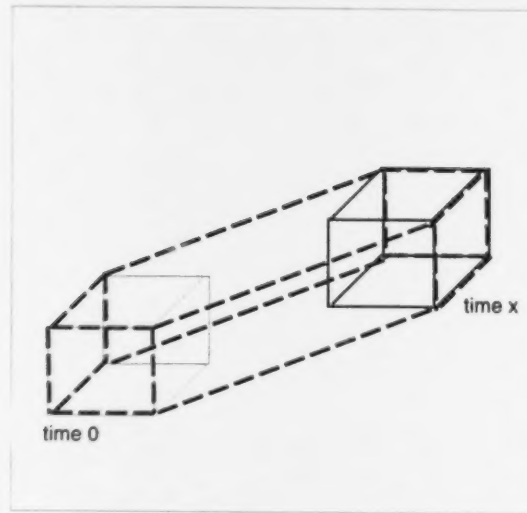


Figure 7: Sweeping a Bounding Box Out Over Time

have a spherical volume much larger than that of the actual object. This causes collisions to be falsely detected, because an object may intersect with the bounding sphere at some point in the future but not with the actual object.

5.4.3 A Bisection Sweeping Approach

By sweeping out both bounding boxes along the direction of motion (as shown in Figure 7) for the desired time interval, we create two new polyhedra. The shape of each swept polyhedron is dependent on the type of motion, both in the number of dimensions and in the direction taken in each dimension. If a bounding box is not moving or is moving only in 1 dimension, the result is another axis-aligned bounding box with 8 vertices and 6 faces, shown in Figure 8. If it is moving in 2-dimensions, the resulting polyhedron will consist of 12 vertices and 8 faces, 2 of which will be bounded by 6 vertices and the rest by 4 vertices, as in Figure 9. If the bounding box is moving in all 3-dimensions, the swept polyhedron will consist of 14 vertices and

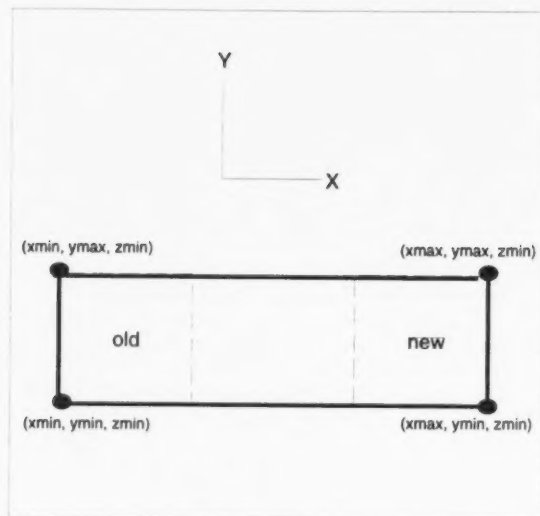


Figure 8: Sweeping a Box In The Positive X-Direction Only (as seen from above)

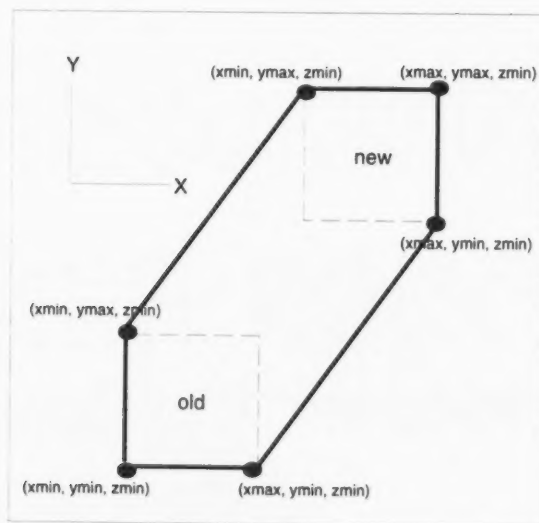


Figure 9: Sweeping a Box In Both Positive X and Y-Direction (as seen from above)

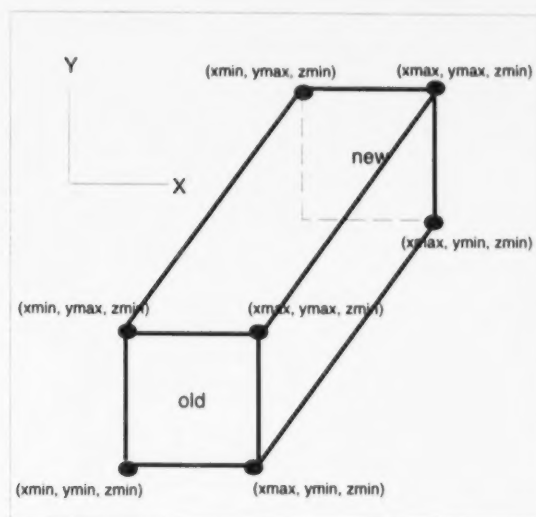


Figure 10: Sweeping a Box In Positive X,Y and Z-Direction (as seen from above)

12 faces, with each face bound by 4 vertices, as shown in Figure 10. The values of the vertices are chosen from the 8 vertices of the original bounding box, and the 8 vertices of the moved bounding box at the end of the time period. The direction of the motion determines which of these vertices are used. For instance, different vertices are used when x and y are both increasing (Figure 9) than when x is increasing and y is decreasing (Figure 11).

For calculating an intersection we use the algorithm presented by Moore and Wilhelms [16], which is applicable to convex polyhedra regardless of the number of vertices or faces. We determine whether polyhedra A and B intersect by: (1) finding if a vertex of B falls within object A ; (2) finding if an edge of B intersects with a face of A ; (3) finding if any of the faces of B are moving through A , by testing if the centroid of each face of B falls within A ; (4) repeating the above steps with A and B reversed. To reduce the number of calculations, we first calculate whether the axis-aligned bounding boxes for these swept volumes intersect before doing the more

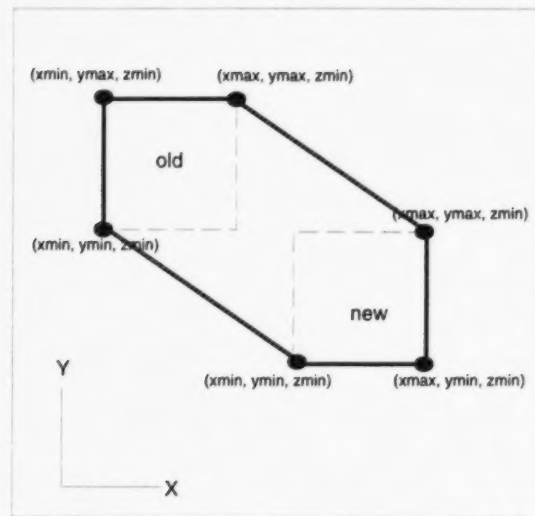


Figure 11: Sweeping a Box In Positive X and Negative Y

expensive polyhedra intersection test. If there is no intersection over the entire swept volume, the two objects do not intersect within that time frame.

The above calculations tell us if the two objects occupy the same space at some point in the future, but not if they occupy it at the same moment. Thus, we use a bisection technique to converge to the actual time of intersection. The time interval is split in two, and the two pairs of polyhedra (those created for the first half time interval versus those created for the second interval) are again tested for intersection. As soon as both tests fail, we can conclude that there is no intersection and stop subdividing. Otherwise, the subdivision continues with progressively smaller candidate polyhedra until the intersection occurs within a time interval of one second. If one of the objects is stationary, no subdivision is required unless we need to determine the exact time of collision - it is enough to know that the collision occurs at some point within the given time frame.

This technique will find non-intersections much more quickly than actual inter-

sections. Thus, it is appropriate in an environment in which many pairs of objects are checked for collision, but few collisions actually occur. This technique is highly accurate, and will detect collisions even if only the corners of boxes intersect.

The above intersection method works only on polyhedra which have a discernable inside and outside, that is, they must be closed. To ensure that this holds for all objects, the three dimensions of an object's bounding box are created with a minimum width. This is merely a change to the internal representation, which does not affect how the object appears when drawn. It is a reasonable assumption, since all objects in the natural world have three dimensions, even though the width in a dimension may be very thin.

5.4.4 Combining a Sphere Approximation with Sweeping

While spheres may be too large to be useful for collision prediction, they provide a fast initial test for collision, as well as a smaller time frame in which the collision could occur. The sphere approximation and the sweeping methods can be combined to produce a method as accurate as the sweeping method alone, but not as time consuming.

For example, say we want to predict a collision t steps in advance. Using the equations given by Dworkin [7] and detailed in Section 2.2, the starting and ending time of the sphere collisions are found. If these times are both negative, the collision has already occurred. If the starting time is greater than t , a collision will not occur in the time of interest. If the starting time is negative, but the ending time is not, then the spheres are currently intersecting. Thus, we use the more accurate sweeping method to check for a collision in the time range $[0 \text{ to } t]$. If the starting time falls within $[0 \text{ to } t]$, we can calculate where the bounding box would be at the starting time, and then only check for a collision from that point to the time $(t - \text{starting\_time})$.

5.5 Extensions for Sub-object Collision Detection

Collision detection at an object level can prohibit natural grouping of objects in an environment. For instance, we would logically create a table object as having four legs and a flat surface. If we want to permit objects to move between the legs, but not through the legs, the current object-level collision detection scheme would be inadequate. Detecting a collision with the bounding box of the table would not allow movement between the table legs, since they fall within the box. Currently, the user would be forced to split up the table into five separate objects and detect collision with each leg separately. Alternatively, the user may wish to trigger different events depending on the part of the object involved in the collision. For these reasons, we need to be able to reference object subparts for collision detection.

Since bounding boxes exist at all levels in the hierarchy, we can use the same approach as used for detecting collisions at an object level to detect collisions at a sub-object level. The user can already apply tags and labels to reference a specific part of an object's geometry. However, we need to allow the user to reference these parts outside of OML code as well, to specify them for collision detection. One approach would be to maintain a list in each object instance containing its labels and pointers to the sub-objects they reference. This list would have to be resolved by the OML interpreter when a new object instance is created.

The *OML_collision* command could then be modified to include not only the two instances, but also the label of the subpart that should be checked for interference. This command would then be:

event = *OML_collision*(*instance1*, *subpart1*, *instance2*, *subpart2*, *time*).

If no subpart was specified, then the entire instance would be used for detecting a collision.

Chapter 6

Culling

6.1 Introduction

Drawing complex objects is very time consuming and hinders the performance of virtual reality applications. Since real-time response is required to give an accurate feel to the application, object complexity, and thus object realism, is usually sacrificed to gain speed. Much effort is wasted drawing objects that cannot be seen given the viewer's current position and line of sight. Culling these invisible objects before drawing could improve the speed of virtual worlds, allowing more complex and thus, more realistic, objects to be used. However, determining which objects are visible can also be expensive, thus, a successful culling method must determine visible objects at a lesser cost than that of actually drawing the objects.

In order to be able to cull at the object level, a new OML drawing command is added which supersedes the GL and PHIGS specific draw commands. Using this OML command, we can first determine whether an object is viewable before the language specific command is called. Four different culling techniques have been implemented, with varying degrees of accuracy and computational complexity. These methods could be used individually or applied in succession, using less costly techniques to remove many of the objects before relying on more expensive tests. The methods are described here, and a discussion of their accuracy follows in Section 7.2.

6.1.1 Calculating the View Volume

The view volume is calculated once for each iteration of the drawing commands. To signal the start of a new drawing round, the user must first call **OML_Init_Draw**, then call **OML_Draw** for each instance to be drawn.

The view volume is approximated as a pyramid, with the narrow end at the current eye-position. The visible range is calculated as 75 degrees in all directions from the line of sight, which is a cautious estimation. Generally, a user would prefer that some time is wasted drawing objects unnecessarily rather than not drawing objects that are actually visible. In experiments, a view volume of 75 degrees captured all visible objects and still culled enough objects to provide an improvement in the polygon drawing rate, as will be seen in Section 7.2. Unfortunately, we do not know how far to extend the view volume along the line of sight, since technically, the user can see for an infinite distance. The extension of the view volume is approximated using ten times the distance from the first object to the eye position - should any of the following objects prove to be further away, the volume is recalculated and this new distance stored for further comparisons. The view volume coordinates are then rotated according to the quaternion associated with the current eye position.

At this point, we also determine the rectangular, axis-aligned bounding box that corresponds to the view volume and the bit flag mapping to the drawing area, using the same techniques used for mapping objects for collision detection in Section 5.3.4.

6.2 Culling Techniques Implemented

6.2.1 A Sphere Approximation

Although axis-aligned bounding boxes allow fast intersection tests with other objects, it is more difficult to correctly detect intersections between a box and a prism/pyramid (the view volume) which is not necessarily axis-aligned. When using a box for culling, it is difficult to know which points to check for intersection. With

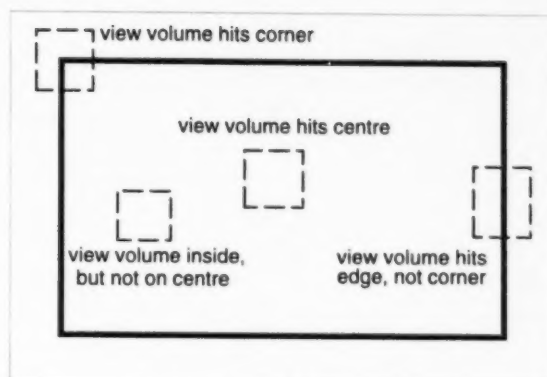


Figure 12: Difficulty in Choosing Test Points for Culling A Bounding Box

a large box, many possible locations for the view volume are possible, as shown in Figure 12.

However, only simple calculations are required to determine if an intersection exists between a cone and a sphere. We can use the values stored for the object's bounding box to create an approximate bounding sphere - the diagonal distance from the minimum coordinates of the box to the maximum coordinates of the box becomes the diameter of the bounding sphere.

We can reject objects that are too small or too far away to be seen by calculating the ratio of the object's diameter to the distance between the viewer and the object. Conversely, objects that surround the viewer can be trivially accepted as being visible. This is the case when the distance from the object's centre to the viewer is less than the radius of the sphere, as shown in Figure 13.

Next, we test for intersection between the object sphere and the viewing cone. An intersection occurs if the angles of the viewing cone overlap with the angles from the line of sight to the outside of the sphere. There are three cases of overlap to consider. In the first, most of the object lies within the view volume, in which case we check if the angle from the line of sight to the object's centre overlaps with the angles of the viewing cone (see Figure 14). In the second case, part of the object, not including the

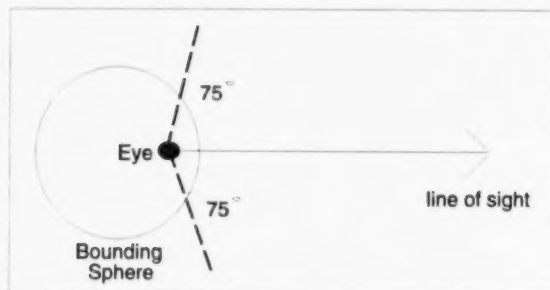


Figure 13: Eye Position Inside Object

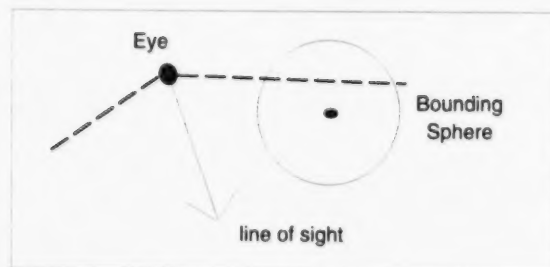


Figure 14: Centre of Object is in Viewer's Range

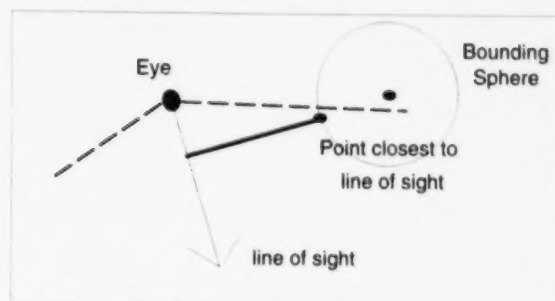


Figure 15: Point on Rim of Object is in Viewer's Range

centre, falls within the view volume, as shown in Figure 15. Here, we check to see if the angles to the outside of the sphere fall within with those of the viewing cone. In the last case, shown in Figure 16, the view volume falls within the sphere, so we test if the angles of the viewing cone fall within the angles to the outside of the sphere. If none of these cases hold, the object is not visible.

Figure 17 will be used to show how these angles are calculated. In this figure, P is the centre of the sphere and r is the radius of the sphere. Q , which lies on the line OT , is the point on the sphere closest to the line of sight, whereas the furthest point lies on the line OS . The angle $\angle PQO$ is 90 degrees. The viewing cone angles are given by $\angle AOX$ and $\angle XOB$, which in our case are each 75 degrees.

We wish to calculate the angle between the line of sight and the points on the sphere closest to and furthest from this line. To do this we use the fact that every point on the outside of the sphere is an equal distance from the centre, and that we can calculate the distance from the eye to the centre of the sphere.

The angle between the centre of the sphere and the point on the sphere closest to the line of sight is calculated as follows:

$$\angle POQ = \arcsin \frac{r}{OP}.$$

Because of the properties of a sphere, this is also equal to the angle between the

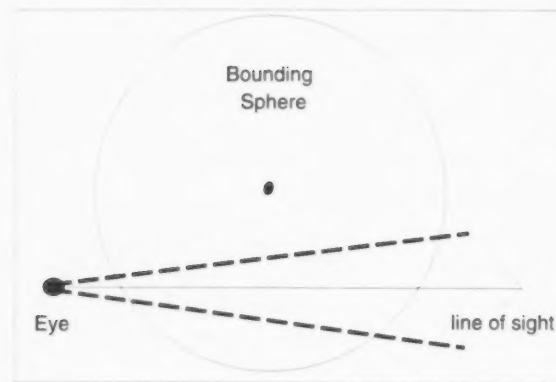


Figure 16: Viewer's Range is Within the Object

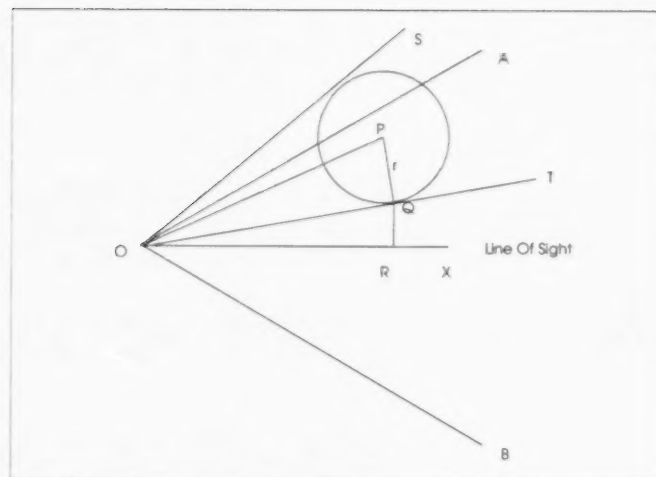


Figure 17: Calculating Angles for Sphere-Cone Culling Method

centre of the sphere and the point on the sphere furthest from the line of sight. Next, the angle between the centre of the sphere and the line of sight is calculated as:

$$\angle POX = \arccos \frac{PO \cdot OX}{||OP||}.$$

Thus, the angles between the line of sight and the points on the sphere are:

$$\begin{aligned}\angle TOX &= \angle POX - \angle POQ \text{ (near point)} \\ \angle SOX &= \angle POX + \angle POQ \text{ (far point)}\end{aligned}$$

The angle range for the view volume is from $\angle XOB$ to $\angle AOX$, whereas the angle range for the sphere is from $\angle TOX$ to $\angle SOX$. The sphere and the view volume intersect if these angle ranges overlap.

If only this culling method is used, we do not need to calculate the view volume, or the bounding box or bitmap flag associated with it. This saves several calculations each drawing iteration.

6.2.2 An Approximate Object/View Volume Intersection

Determining the visibility of an object can be seen as a test for intersection between the view volume and the object space. This can be quickly checked using the same method of comparing bit flags indicating spatial location as done for collision detection, that is, we AND the object's flag with that calculated for the view volume. However, instead of further subdividing the space, we simply reject those objects that do not intersect with the view volume at all, indicated by an AND-value of 0.

Similarly, we can use the bounding box collision technique check for overlap between the axis-aligned approximations for the view volume and the object. This will remove more objects than checking the bit flags, but requires at most six floating point comparisons, whereas checking the bit flags is one AND operation.

6.2.3 An Exact Object/View Volume Intersection

The view volume is a five-sided prism, so we can do an exact object and vol-

ume intersection test by using the polyhedra intersection test implemented to test whether objects will collide at some future point, described in Section 5.4. However, this is a computationally expensive method, and is inappropriate for culling in most applications (unless extremely complex objects are used).

6.3 Extensions for Culling Sub-Objects

The above techniques all cull at the object level. However, for large and complex objects, it might be advantageous to more finely determine which parts are visible.

First, we would need to find a fast heuristic to decide the appropriateness of finer culling for any given object. This might be the number of sub-objects involved, the size of the object, the percentage of the object within the viewing volume or a combination of these factors.

Currently, the language-specific drawing structure of OML objects is stored and updated only when that object changes. If we are to cull sub-objects, we would need to traverse the object each time and flag the parts that are visible, instead of merely drawing the existing structure. As well, we would have to apply all transformations along the hierarchy whether or not that part of the object is drawn. This may be more time-consuming than just using the existing structure.

6.4 Limitations of These Techniques

These methods remove objects that are behind the user or at too sharp an angle or too distant to be seen, however, they do not identify objects that are obstructed by other objects. Intuitively, we feel removing hidden objects might be too computationally expensive to meet the real-time objectives of our virtual reality system, however, no work in this area has been attempted.

Chapter 7

Testing

A common measurement of virtual reality systems is the number of polygons drawn per second. The higher this rate, the more realistic the motion in the application. Although items are culled on an object level, we calculate the polygon rate using the number of polygons contained in the object. A complex object will contain many more polygons than a simple one, thus culling a complex object will improve the polygon rate more than removing a simple one. When calculating the polygon rate, we can use either the total real time, which includes the time spent in I/O and swapping, or the total time spent executing only our process, taken from rusage "user" statistics. The real time rate reflects what is seen by the viewer as the application is running, including slow downs caused by other users or processes. The user time rate does not measure time spent outside our process, and so better indicates the success or weakness of our various algorithms and implementations. In several of the tables used in this chapter, both statistics will be given. All tests were run on a Silicon Graphics 4D/310VGX.

7.1 Evaluating Subdivision in Collision Detection

A major concern in collision detection is that potentially all pairs of objects will need to be checked for interference. In Section 5.2, we propose a subdivision technique

Time Used	# Objects	No Spatial Division (polygons/sec)	One-Level Grid (polygons/sec)	Deeper Subdivision (polygons/sec)
Real	10	13650	13636	13754
Real	20	13426	13867	13564
Real	30	11548	11714	10931
Real	40	10186	10531	10108
User	10	20825	19310	18708
User	20	17634	17500	17608
User	30	16615	16999	16448
User	40	16013	15950	15561

Table 7.1: Polygon Rates - Several Moving, Few Stationary Objects

that maps the object's bounding box to grid locations in the drawing area, giving a position flag that can be ANDed with that of other objects to quickly reject non-colliding pairs.

To test the efficiency of such an approach, we looked at two cases. In the first, a room was created that contained few static objects, and various numbers of moving objects. Each time an object moves, its bit flag, indicating the grid locations intersected, must be updated. However, only a small number of tests for collision must be carried out for each moving object. In the second test, a maze consisting of many static objects and very few moving ones was created. Here, little work is spent in updating the bit flags, however, many more collision tests are required.

7.1.1 Case of Several Moving Objects, Few Static Objects

Table 7.1 and Graphs 18 and 19 show the polygon rate in tests with various numbers of moving objects. Each moving object checks for collision with four static objects. We tested three variations: using no spatial division, so no time is spent updating the grid; using one level of division, so the grid is updated but no further subdivision occurs; and using the deeper subdivision technique described in Section 5.3.3. These rates were obtained using no culling techniques.

As is to be expected, the overall performance decreased as the number of moving objects increased, regardless of the spatial division technique used. This is due to the

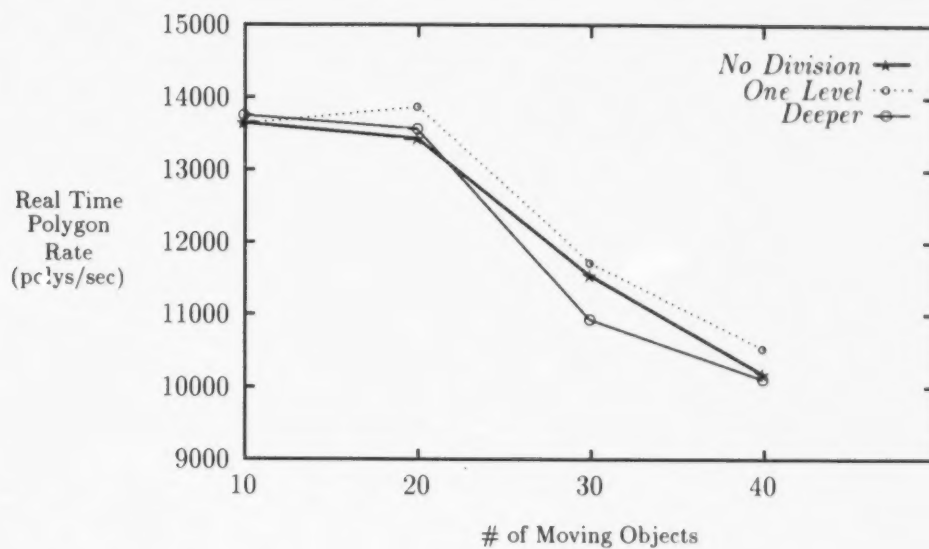


Figure 18: Graph of Real Time Polygon Rates for Several Moving Objects

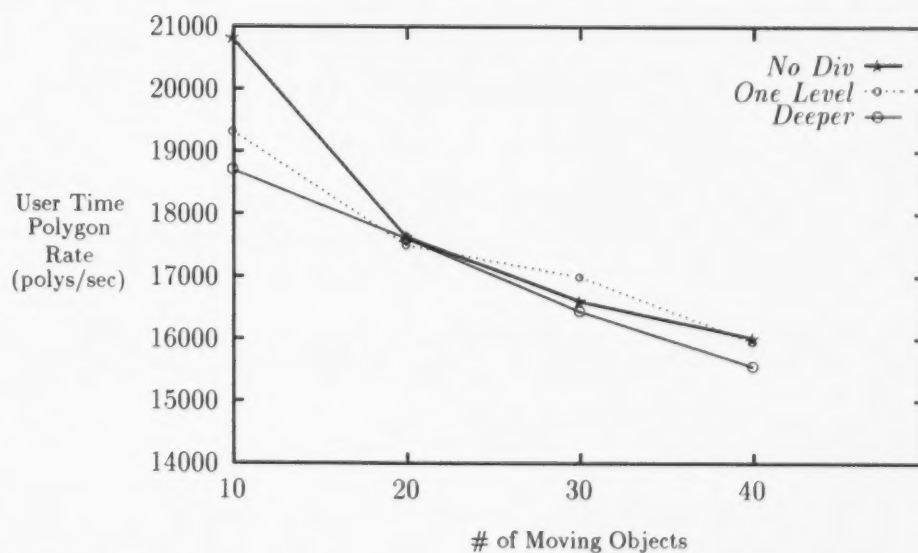


Figure 19: Graph of User Time Polygon Rates for Several Moving Objects

Time Used	# Objects	No Spatial Division (polygons/sec)	One-Level Grid (polygons/sec)	Deeper Subdivision (polygons/sec)
Real	1	19977	20123	18374
Real	5	16182	16192	13896
Real	10	12638	12865	10531
User	1	30882	33244	29342
User	5	23851	24428	19196
User	10	18845	19655	14694

Table 7.2: Polygon Rate - Many Stationary, Few Moving Objects

greater amount of time required for collision detection and for processing behaviours. It can be seen that neither of the division techniques provided a clear advantage for detecting collisions in this environment. Apparently, the time needed to constantly update the bit flag for the moving objects exceeded any time gained by quickly rejecting non-collisions.

7.1.2 Case of Few Moving Objects, Several Static Objects

Table 7.2 shows the polygon rate for tests with 600 stationary objects, and small numbers of moving objects. Each moving object must check for collision with all 600 static objects. The timings are for 1000 redraws of the scene, with no culling. These results are also presented in Graphs 20 and 21.

The polygon rates in this set of tests are generally higher than those of the previous section. Even though more objects are drawn, fewer objects are moving, thus fewer behaviours need to be updated. Processing behaviours can be quite expensive, causing the application to be slower.

In these tests, it can be seen that using deeper subdivision does not provide any gain in the polygon rate. There is a very slight improvement in comparing the bit flags over using the collision method with no spatial division. Generally, interesting virtual environments will contain many more moving objects than we have used here, which, as shown in the previous set of tests, will reduce the efficiency of our subdivision methods.

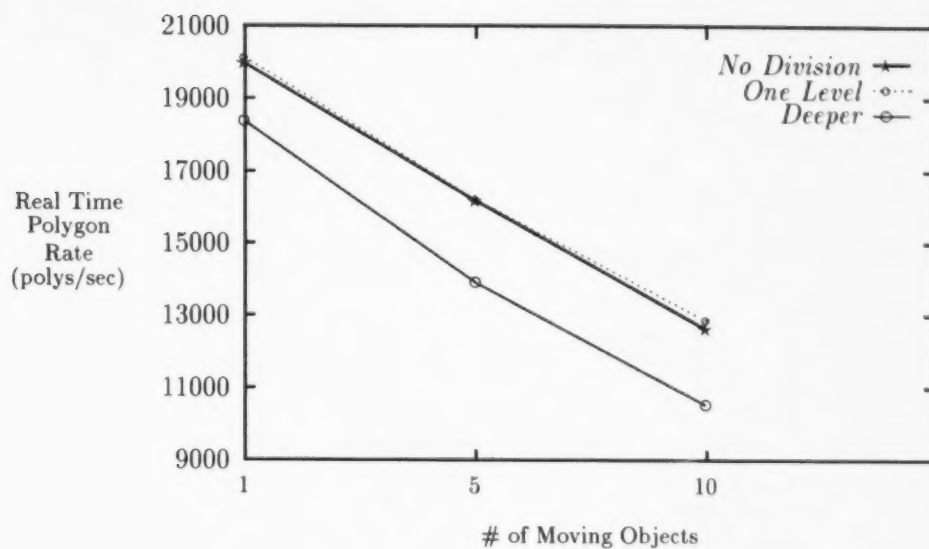


Figure 20: Graph of Real Time Polygon Rates for Several Stationary Objects

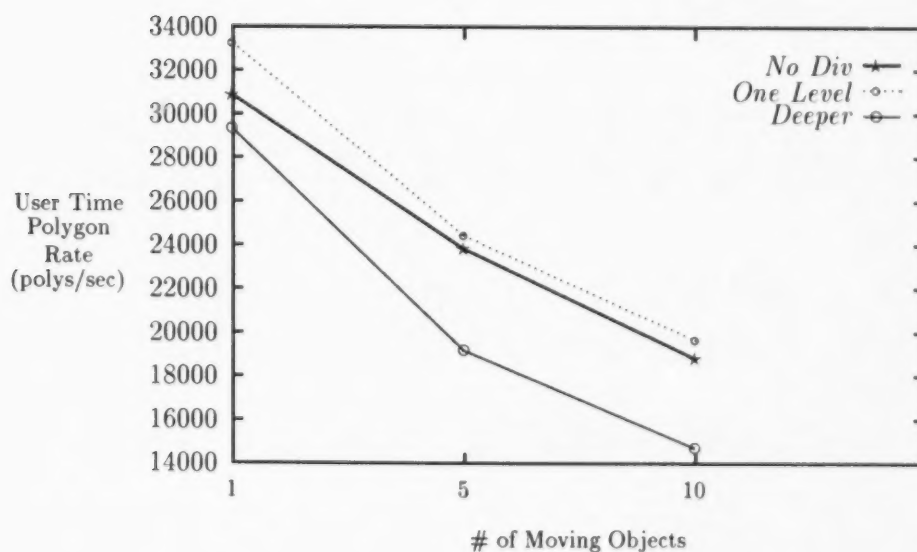


Figure 21: Graph of User Time Polygon Rates for Several Stationary Objects

7.1.3 Analysis

The subdivision technique used to reduce the number of checks for collision does not appear successful in its current implementation, unless extremely large numbers of objects are involved. For environments in which several objects are moving, too much time is spent updating the bit flags. Even ignoring the update time, we must still traverse all possible pairs of collisions, and there does not appear to be enough time savings in comparing two flags over comparing the (at most) six floating point numbers which represent the bounding boxes. A more successful method would be one that actually reduces the number of pairs to be tested instead of simply changing the accuracy and speed of the testing method.

7.2 Comparison of Culling Techniques

7.2.1 The First Test Environment

To test the effect of culling, we can compare the number of polygons drawn and the time required to traverse a given path through an environment with and without culling. Both the motion and the line of sight must be the same for the two tests, since it is the line of sight that determines which objects are visible.

A maze consisting of a square floor and 81 separate boxed wall segments was created using **JDCAD**¹ to test our four culling techniques. Each wall piece is a separate object to allow culling at the object level. The floor is 36.6 metres square, and 0.3 metres thick. Each wall segment is a box of dimensions 4.5 X 4.5 X 0.3 metres, thus each object consists of 6 polygons.

The eye position of the user begins at a corner of the maze, and travels towards the centre of the maze in a spiral manner. Each turn is made in increments of 6 degrees. The eye travels along the x and y axes in steps of 0.5 metres per time step,

¹JDCAD is a Computer Aided Design tool developed at the University of Alberta by Jiandong Liang, which allows the user to design hierarchical objects using a six-dimensional input device. This model is then converted into OML code.

	Max. # Objects Any One Iteration	Average # Over All Iterations	Average %age Over All Iterations
Culled by Sphere Test	70	40	48
Culled by Bitmap Test	60	21	25
Culled by Bounds Test	71	28	34
Culled by Volume Test	75	43	52
Actually Drawn	65	39	48

Table 7.3: Axis-Aligned Culling Test For 82 Objects

at a height of 2 metres. At this speed, 686 redraws of the scene are required, giving a total of 3380040 polygons drawn. Slight variations in the path length, causing more redraws, did not greatly vary the results. At no point in the test does the eye position travel outside the maze. (Having the eye outside the maze, that is, looking into blank space, would increase the number of objects culled, but this does not occur in typical applications).

7.2.2 Calculating the Number of Objects Culled

In the first set of tests, the maze pieces were axis-aligned, thus simulating the bounding boxes of OML objects. More substantial time savings would be seen if each wall piece was actually a complex, hierarchical object, which is more expensive to draw than a box. Table 7.3 shows the maximum and average number of the 82 objects culled by each of the four methods, as well as the number actually drawn.

These tests show the view volume/object intersection to be the most accurate, followed by the sphere/cone test, the intersection of the bounding boxes, and the intersection of the bit flags. The objects identified for culling by the exact view volume/object intersection method is a superset of those identified by the other techniques. The sphere/cone intersection method culls all of the same objects that are identified by the bit tests, and misses only a few that are culled by the bounding box intersection test. In this experiment, at most 7 objects of the 82 that were culled by the bounding box test were missed by the sphere test. (These misses would most likely occur because a bounding sphere is a larger approximation of the object, and

	Max. # Objects Any One Iteration	Average # Over All Iterations	Average %age Over All Iterations
Culled by Bitmap Test	60	21	25
Culled by Bounds Test	24	7	9
Culled by Sphere Test	62	12	15
Culled by Volume Test	6	2	3
Actually Drawn	65	39	48

Table 7.4: Culling Tests Applied in Order of Complexity For 82 Axis-Aligned Objects

so is more likely to be within the visible range). Similarly, the bounding box test identifies the same objects as the bitmap test, but misses several of those identified for culling by the sphere test. The bounding box test basically removes the objects behind the viewer, but not those at too sharp an angle to be seen, which can be culled by the sphere test. The bitmap test is the least accurate, but it is the fastest to apply, and can be used to quickly reduce the number of objects to be tested.

The accuracy of these culling techniques corresponds to the computational complexity required to implement them. We next applied the culling techniques in terms of increasing accuracy (and increasing complexity). If an object was identified for culling by one method, it was not checked by the other methods. Thus, objects that are far from the view volume can be quickly and inexpensively culled. However, objects that are in or near the view volume must go through all four culling tests. As shown in Table 7.4, few polygons were culled by the most time-consuming method, the view volume/object intersection test. We must thus consider whether the gains made by culling these few objects outweighs the cost of applying this test (the answer will become obvious in Section 7.2.3).

In the second set of tests, the maze and all its pieces were rotated by 45 degrees, thus making them non-aligned with the x and y axes. This resulted in a less tight fit by the bounding boxes, with the bounding volume larger than the original object. It is thus more likely to intersect with the view volume causing the object to be drawn even though it may not actually be visible. The culling values obtained from the rotated maze, given in Table 7.5, show that the intersection tests using the bit flag

	Max.# Objects Any One Iteration	Average # Over All Iterations	Average %age Over All Iterations
Culled by Sphere Test	70	39	48
Culled by Bitmap Test	32	2	2
Culled by Bounds Test	37	4	5
Culled by Volume Test	73	42	51
Actually Drawn	66	40	49

Table 7.5: Unaligned Culling Test For 82 Objects

	Max# Objects Any One Iteration	Average # Over All Iterations	Average %age Over All Iterations
Culled by Bitmap Test	32	2	2
Culled by Bounds Test	29	2	3
Culled by Sphere Test	70	35	43
Culled by Volume Test	5	2	3
Actually Drawn	66	40	49

Table 7.6: Culling Tests Applied in Order of Complexity For 82 Unaligned Objects

and the bounding box are far less useful in culling objects than in the aligned object tests. However, the sphere/cone method is still quite successful in detecting objects to cull. When the tests are applied in succession, as shown in Table 7.6, many more objects are removed by the sphere test than when the objects were axis-aligned, since fewer were culled by the earlier bit flag and bounding box tests.

In the rotated maze, the bounding boxes of objects are a less accurate representation of the underlying geometry. The bounding box and bit flag representing the eye will also be less accurate. The eye will be moving at an angle, so the axis-aligned bounding box representation of the view volume will contain much more area, including area potentially behind the viewer. Moving at an angle does not affect the cone representation of the view volume, however, thus the number of objects culled by the sphere/cone method in the rotated maze is similar to that of the axis-aligned maze.

Next, we created a maze consisting of three axis-aligned floors, each floor having the same configuration as in the tests described previously. The separation distance between these floors had a great effect on the number of objects culled. Since no

	Max. # Objects Any One Iteration	Average # Over All Iterations	Average %age Over All Iterations
Culled by Bitmap Test	180	62	25
Culled by Bounds Test	72	21	9
Culled by Sphere Test	226	109	44
Culled by Volume Test	22	5	2
Actually Drawn	129	48	20

Table 7.7: Culling Tests Ordered by Complexity For a Maze of 3 Separated Floors

hidden surface removal is attempted, having three floors directly on top of each other produces numbers similar to those for one floor, as all floors are within the visible range (75 degrees in all directions). However, separating the floors by a large amount so that not all floors are visible shows that the culling methods remove a large percentage of the objects in the environment. (For our test, a separation distance of 80 metres was used.) This test corresponds to large environments in which only a small number of objects can actually be seen. Table 7.7 shows the percentage of the 246 objects removed by the series of culling techniques. Again, the greatest gain in the number of objects culled is shown by the sphere/cone intersection method.

When using the bounding box and/or the bit flag intersection techniques, little difference in the polygon rate was seen between the one and three floor models. Both these techniques basically remove half the drawing area, the part that is behind the current eye-position, since the bounding distance in front of the eye extends infinitely. Therefore, whether the maze consists of only one floor, or of several floors, the corresponding objects on each level will be culled and will be drawn (assuming that the floors are identical and are aligned, as in our test).

Approximating the view-volume as a cone, however, will cull many more objects on other floors. Not only will the objects behind the eye be removed, but many objects close to the eye-position in the x and y dimensions, but on different floors, will be culled as well, since they are outside the 75 degree range of the eye. Thus, if the eye is on the first floor, more objects on the second floor will be outside of the angle, and even more on the third floor.

	Time (sec)	Polygon Rate (polygons/sec)	Speedup Factor
No Culling Technique	351 (real)	9624	-
All 4 Culling Methods	464 (real)	7286	0.76
Bitmap, Bounds, Sphere	294 (real)	11480	1.19
Bitmap, Bounds	308 (real)	10968	1.14
Sphere	283 (real)	11949	1.24
No Culling Technique	94 (user)	35824	-
All 4 Culling Methods	351 (user)	9643	0.27
Bitmap, Bounds, Sphere	74 (user)	45987	1.28
Bitmap, Bounds	80 (user)	42093	1.18
Sphere	70 (user)	48101	1.34

Table 7.8: Timings of Culling For 10 iterations of a 1-Floor Maze

7.2.3 Calculating the Improvement in Speed

Although the techniques can all be shown to allow some culling of objects, we must also show that there is a time benefit to do so over all applications. For instance, even though it might be appropriate to cull very complex objects using a computationally expensive method, this would be inefficient in most applications when users create only simple objects.

In Table 7.8, we compare the time required for 10 runs through the one-floor maze applying various combinations of the culling methods. As suspected, applying the expensive but accurate view volume/object intersection test proved inadequate as a general culling technique, in fact slowing down the application, as indicated in the Graphs 22 and 23. The sphere/cone method, however, seems to be quite promising, giving slight speedups on the one-floor maze, and larger speedups on the three-floor maze (see Table 7.9 and Graphs 24 and 25). Adding the tests for intersection of bit flags and bounding boxes further increased the polygon rate, but not considerably.

7.2.4 The Second Test Environment

The objects in the first environment are static and fairly simple. To further test our culling techniques, we wanted an environment which contained more complex

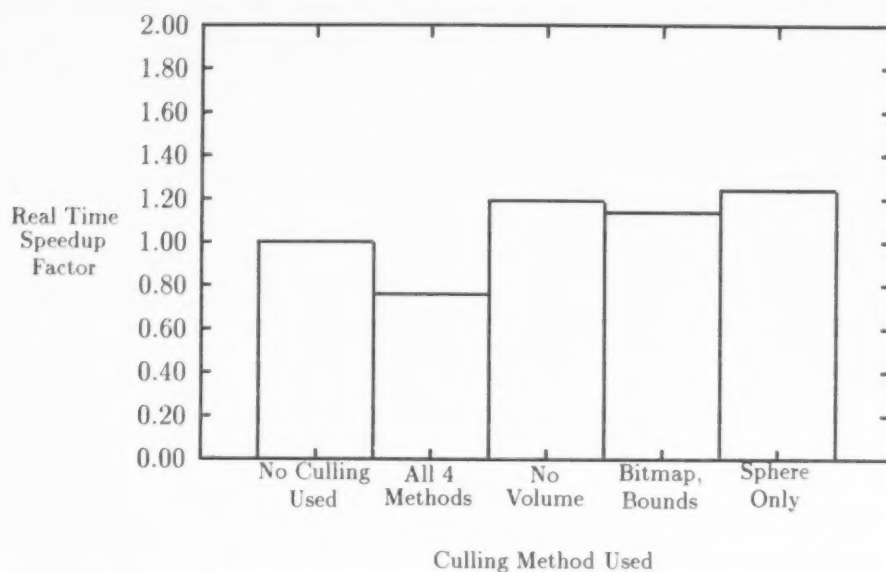


Figure 22: Graph Indicating Real Time Speedup of Culling Techniques for 1 Floor

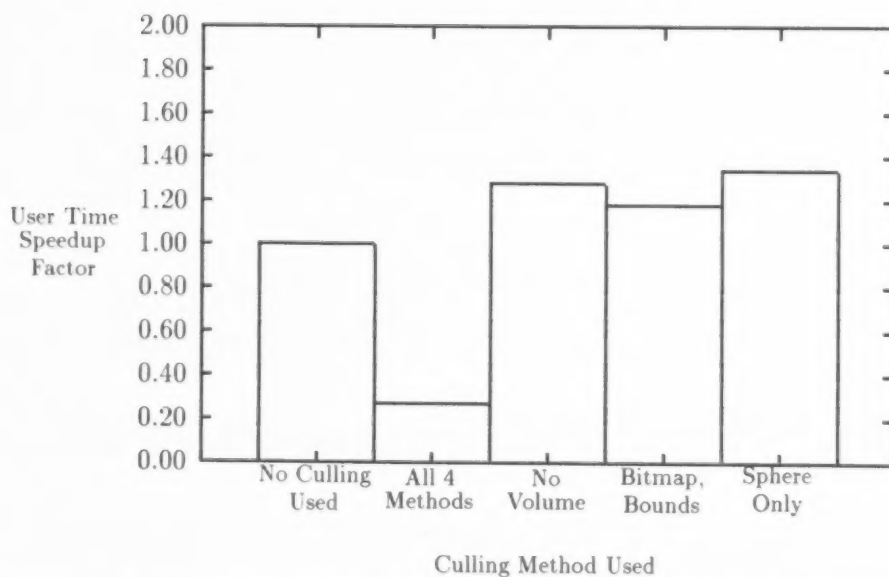


Figure 23: Graph Indicating User Time Speedup of Culling Techniques for 1 Floor

	Time (sec) (sec)	Polygon Rate (polygons/sec)	Speedup Factor
No Culling Techniques	903 (real)	16852	-
All 4 Culling Methods	925 (real)	16435	0.98
Bitmap, Bounds, Sphere	516 (real)	29456	1.75
Bitmap, Bounds	712 (real)	21373	1.27
Sphere	509 (real)	29879	1.77
No Culling Techniques	401 (user)	37944	-
All 4 Culling Methods	754 (user)	20165	0.53
Bitmap, Bounds, Sphere	207 (user)	73497	1.94
Bitmap, Bounds	338 (user)	44941	1.18
Sphere	213 (user)	71342	1.88

Table 7.9: Timings of Culling For 5 iterations of a 3-Floor Maze

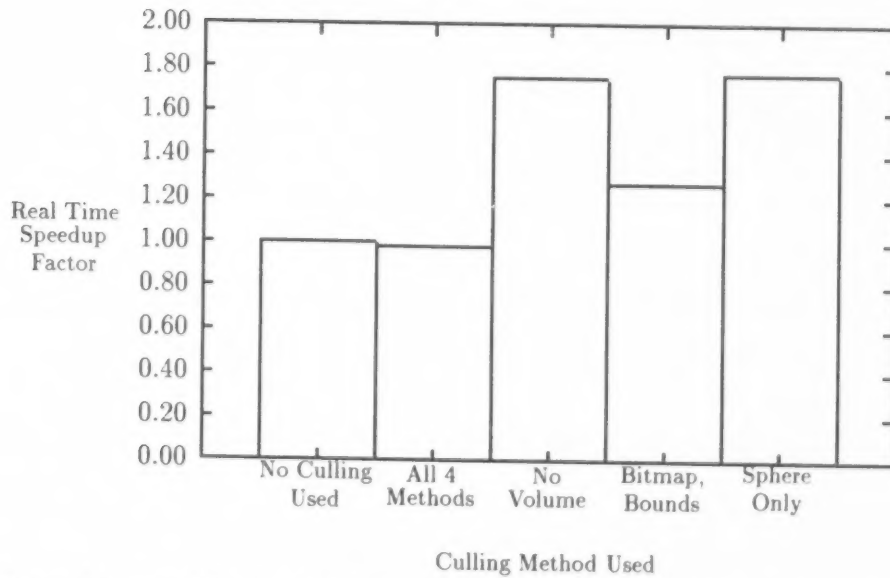


Figure 24: Graph Indicating Real Time Speedup of Culling Techniques for 3 Floors

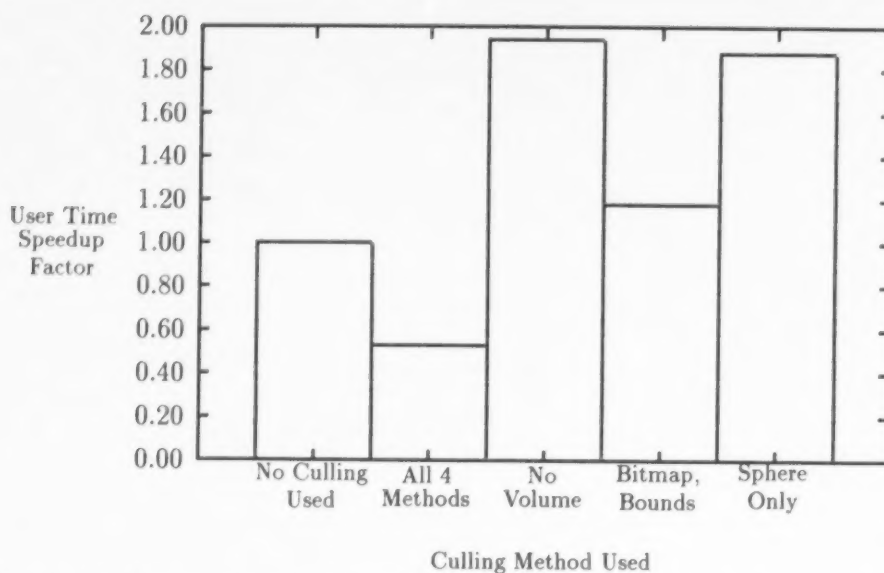


Figure 25: Graph Indicating User Time Speedup of Culling Techniques for 3 Floors

objects. We created a room, consisting of some static objects, with several smaller objects constantly moving throughout it. Each such moving object consists of 100 polygons. The eye position is fixed in the centre of the room, and rotates around the room, 6 degrees at a time. At this speed, 60 redraws are required per complete rotation.

The first column in Table 7.10 shows the average percentage of the 2380 polygons culled per redraw by each method applied individually, whereas the second column shows the average percentage culled when the methods are applied sequentially. As

	Applied Individually	Applied Successively
Culled by Bitmap Test	9%	9%
Culled by Bounds Test	15%	6%
Culled by Sphere Test	50%	35%
Culled by Volume Test	52%	2%
Actually Drawn	48%	48%

Table 7.10: Percentage of Polygons Culled

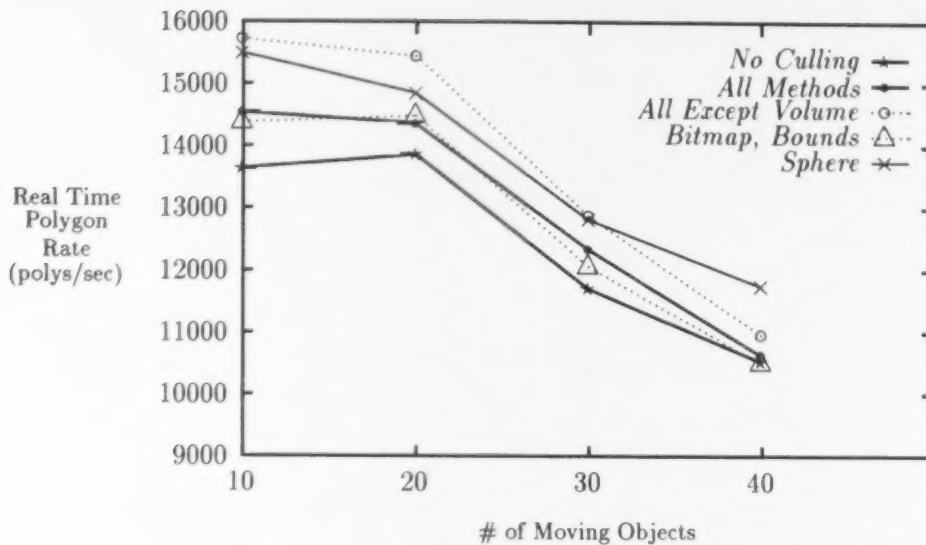


Figure 26: Graph of Real Time Polygon Rates for Culling Techniques

with the first test environment, only a small percentage of polygons that can be culled by the view volume test are missed by the sphere/cone method.

Tables 7.11 and 7.12 list the polygon rate for various combinations of the culling techniques, using real-time and process-time respectively. These values are also presented graphically in Figures 26 and 27. The left-most column in each table indicates the number of moving objects used in the test. The larger the number of moving objects, the slower the polygon rate, because many more objects must have their behaviours updated, and must be checked for collision.

It can be seen that the tests combining bitmap flag, bounding box and sphere/cone culling methods showed the most improvement in polygon rate. Adding the view volume test produced rates higher than those in the maze environment, but they were still lower than the combination of bitmap/bounding box/sphere, or the sphere test alone.

# Moving Objects	No Culling (polys/sec)	All Methods (polys/sec)	All Except Vol. (polys/sec)	Bitmap, Bounds (polys/sec)	Sphere (polys/sec)
10	13636	14542	15729	14380	15500
20	13867	14358	15438	14480	14844
30	11714	12343	12878	12058	12819
40	10531	10641	10962	10499	11746

Table 7.11: Real-Time Polygon Rate With Various Numbers of Moving Objects

# Moving Objects	No Culling (polys/sec)	All Methods (polys/sec)	All Except Vol. (polys/sec)	Bitmap, Bounds (polys/sec)	Sphere (polys/sec)
10	19310	19501	22080	20254	21963
20	17500	17726	19994	18224	18874
30	16999	16835	18757	17453	18036
40	15950	15385	18242	16030	17702

Table 7.12: User-Time Polygon Rate With Various Numbers of Moving Objects

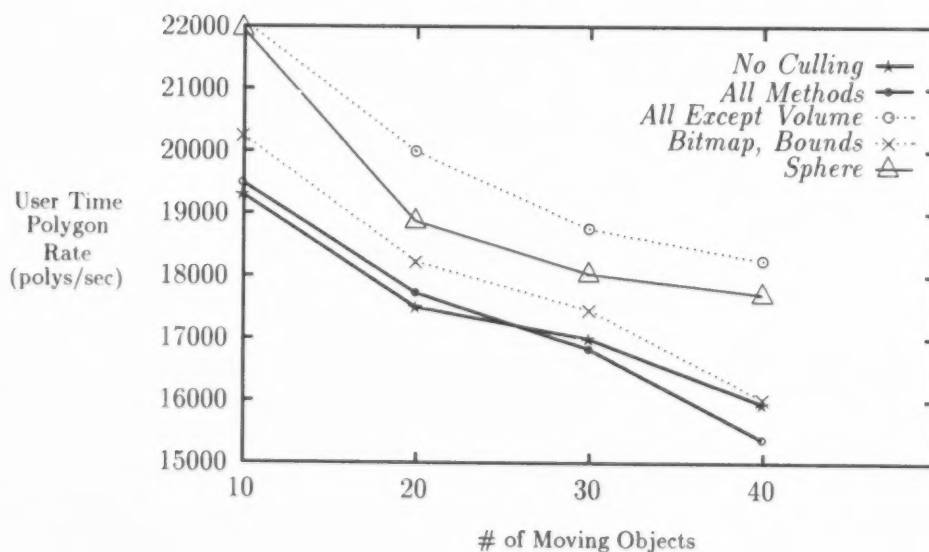


Figure 27: Graph of User Time Polygon Rates for Culling Techniques

Chapter 8

Conclusions and Future Work

8.1 Conclusions

Bounding boxes provide a reasonable approximation to the underlying geometry of an object for most applications. Their use allows collision detection calculations to be greatly simplified, adequately meeting the constraints of a real-time virtual reality system. The hierarchical method of building the bounding box facilitates the future extension to providing collision detection between sub-objects, and the culling of subparts.

The subdivision technique used to reduce the number of checks for collision does not appear successful in its current implementation, unless a large number of objects (more than the 800 we used) are involved. New methods should be investigated that reduce the number of object pairs tested.

The sphere-cone culling method proved the most successful. In some cases, the bit flag and bounding box culling methods caused some further improvement. Often, however, the cost of constantly recalculating the view volume to allow these tests negated these benefits. If the view volume were to rarely change, reducing the number of recalculations, the bounding box method could be useful. Since the use of bit flags does not provide a clear improvement to collision detection, it is not worthwhile to implement it simply for culling purposes.

The accuracy required of the intersection test and collision prediction varies with the needs of the user. As well, some applications might benefit from a slow but accurate culling technique such as the exact view volume/object intersection method, whereas most would not. We do not want to impose a method that would potentially hinder any application, however, we do not want to force the user to write excessive amounts of code to reimplement features OML claims to have.

This leads us to conclude that the user should have greater control over the internal OML methods used for collision detection and culling. As well, work should be done to categorize the characteristics of environments, and to match the different OML methods with these characteristics. This could be provided as a guide to the user to help in maximizing the efficiency of OML applications.

8.2 Future Work

8.2.1 Extensions to OML for Collision Detection

Currently, the user must explicitly specify all pairs between which collisions should be monitored. This will become tedious in complex environments consisting of many objects. Commands should be added to allow an event to be signalled on a collision with any object, and for the negation, a collision with no object.

The appropriateness of a collision prediction method is dependent on the particular virtual reality application, both in terms of the required accuracy and the effort spent in calculation. In the current implementation, the user can simply choose to not use the existing collision detection scheme. Thus, if accuracy is a requirement for an application, the user can implement this themselves. More control could be given to the user to choose the method used for collision prediction from a given set of methods, instead of merely allowing them to add their own code to override the default we choose. Additionally, different methods could be used for different pairs of objects, allowing the user to prioritize where the time for calculations should be spent.

In some test cases, the speed of the moving objects caused collisions to be missed, since the object was in front of the boundary at one time step, and on the other side the next. We countered this simply by making the boundaries thicker. The problem could be addressed more generally by adding a new detection method that would sweep the object by one time step, forming a new polyhedron in the same manner used for collision prediction. These swept volumes representing the two objects could then be compared for intersection using the already encoded polyhedra collision detection method.

As well, an even more exact collision detection technique could be implemented using the objects' underlying vertices once the bounding boxes intersect. Again, this could be done using the current polyhedra intersection technique. The user would be spared implementing this themselves.

The code permitting collision detection between sub-objects should also be added. Currently, the user must split an object into smaller pieces if collision detection is needed for any of the individual subparts. Being able to access subparts for collision detection would greatly enhance OML's flexibility.

8.2.2 Improvements to Collision Detection

It is wasteful to check for a collision between two objects at each time step. An alternative to using a spatial grid would be to determine the time until a collision is first possible using the objects' velocities and the distance between the bounding spheres, and then to check for a closer intersection at each time step from then on. We would only need to recalculate the time until an initial collision if the velocity of either object changes. When recalculating an object's velocity, the old velocity could be stored along with the new one, which could be compared when processing the collision list to see if the earliest collision time must be recalculated.

The current attempt to reduce the all-pairs problem in collision detection by subdividing the drawing area appears unsuccessful for most applications. Perhaps an alternative approach could be used when many pairs must be checked. Specifically,

the drawing area could be divided in the current manner, but each grid section would maintain a list of all objects that occur in it. Then collisions would only have to be checked with other objects within that grid location. The current method of having a linked list of pairs of objects to check would have to be changed to accommodate this. Additionally, we need a method of determining that object *A* is within the same grid section as object *B*, and that *A* is also within *B*'s list of objects with which to detect interference. The subdivision technique will only be successful if this searching method is very fast and efficient - a naive search through a linked list will be inadequate. It must also be ensured that an event is not signalled twice just because a pair of objects intersects in more than one grid section.

8.2.3 Improvements to Culling

Currently, the view volume, and its corresponding bounding box and bitmap flag, is recalculated each drawing iteration. This is wasteful if the view volume has not changed, or if the change is minimal. Instead, only a significant change in the view volume (one that could bring new items into view) should cause the view volume to be recalculated.

Culling is done only at an object level. Success of this approach is dependent on the user's choice of objects. A bad choice, such as combining all the walls in a multi-floored maze into one object, would get no benefit from our culling techniques, since some section of all walls would be visible at all times. The decision of when to attempt partial culling could be made by either the user or the system. The user, having designed the objects, may be the best judge of whether subpart culling would be worthwhile. However, if we wish to free the user from such decisions, a heuristic must be found which can make these choices based on known characteristics of the object. These characteristics may include the size of the object, or the number of polygons within it.

Bibliography

- [1] W.H. Beyer, editor. *CRC Standard Mathematical Tables*. CRC Press, Inc., Boca Raton, Florida, 26th edition, 1981.
- [2] J.W. Boyse. Interference detection among solids and surfaces. *Communications of the ACM*, 22(1):3-9, 1979.
- [3] S.A. Cameron. A study of the clash detection problem in robotics. *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 488-493, 1985.
- [4] S.A. Cameron. Collision detection by four-dimensional intersection testing. *IEEE Transactions on Robotics and Automation*, 6(3):291-302, 1990.
- [5] J.F. Canny. Collision detection for moving polyhedra. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 200-209, March 1986.
- [6] R.K. Culley and K.G. Kempf. A collision detection algorithm based on velocity and distance bounds. *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1064-1069, 1986.
- [7] P. Dworkin and D. Zelzer. A new model for efficient dynamic simulation. *Proceedings of the Fourth Eurographics Workshop on Animation and Simulation*, pages 135-145, 1993.
- [8] J.D. Foley, A. van Dam, S.K. Feiner and J.F. Hughes. *Computer Graphics - Principles and Practice*. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 2nd edition, 1990.
- [9] A.S. Glassner, editor. *Graphics Gems*. Academic Press, Inc., San Diego, California, 1990.
- [10] M. Green. *Object Modeling Language (OML) Programmer's Manual*. Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, 1993.
- [11] J.K. Hahn. Realistic animation of rigid bodies. *Computer Graphics*, 22(4):299-308, 1988.

- [12] P.M. Hubbard. Interactive collision detection. *Proceedings 1993 Symposium on Research Frontiers in Virtual Reality*, pages 24–31, October 1993.
- [13] D.A. Joseph and W.H. Plantinga. Efficient algorithms for polyhedron collision detection. Technical Report 738, University of Wisconsin, Department of Computer Sciences, Madison, Wisconsin, December 1987.
- [14] M.C. Lin and J.F. Canny. A fast algorithm for incremental distance calculation. *Proceedings of the IEEE International Conference on Robotics and Automation*, pages 1008–1014, April 1991.
- [15] W. Meyer. Distances between boxes: Applications to collision detection and clipping. *Proceedings of the IEEE International Conference on Robotics and Automation*, 1986.
- [16] M. Moore and J. Wilhelms. Collision detection and response for computer animation. *Computer Graphics*, 22(4):289–298, 1988.
- [17] J. Youn and K. Wohn. Realtime collision detection for virtual reality applications. *IEEE Transactions on Robotics and Automation*, pages 415–421, 1993.
- [18] M.J. Zyda, W.D. Osborne, D.R. Pratt and J.G. Monahan. NPSNET: Real-time collision detection and response. *The Journal of Visualization and Computer Animation*, 4:13–24, 1992.